DISS. ETH. NO. 20692

# Integrating Cloud Applications with Mobile Devices

## Design Principles and Performance Optimizations

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Sciences

presented by
IOANA GIURGIU
Eng. Dipl. in Computer Science, TU Cluj-Napoca
born 10 March, 1985
citizen of Romania

accepted on the recommendation of

Prof. Dr. Gustavo Alonso, examiner
Prof. Dr. Timothy Roscoe, co-examiner
Prof. Dr. Srdjan Capkun, co-examiner
Prof. Dr. Christian F. Tschudin, co-examiner

2012

*In loving memory of my father.*

# Abstract

With the growth in popularity of mobile devices and the increase in processing power and connectivity given by current trends, users demand ever more ubiquitous and complex applications. They use their devices much like they use their personal desktops and expect similar levels of capabilities, while maintaining or even increasing battery life. Applications such as video and audio processing, 3D gaming or augmented reality are stretching the resources of current mobile devices, and user experience rapidly degrades when several such applications are run concurrently. In the opposite scenario where applications are hosted and run in the cloud, mobile devices act only as access points. Such an approach, however, often raises usability issues due to varying latency and cannot adapt dynamically to more powerful devices.

This thesis contributes towards overcoming such limitations by proposing a flexible approach to integrating cloud-based applications with mobile devices. Our premise is that by exploring the existing space between the two traditional alternatives, parts of the computing process of applications can be optimally outsourced from the cloud to mobile devices. By doing so, users benefit from the seamless integration between their devices and the cloud high performance. Running complex operations within the cloud reduces the cost of mobile computing and allows even low-entry models to take advantage of the cloud capabilities. Thus, users experience a plethora of unique features enhancing their devices and are able to interact with a wider range of rich applications. With the accelerated increase in application complexity, however, the cloud-located application parts frequently require multiple networked virtual instances. The distribution of these instances on the physical infrastructure impacts the way users are able to interact with their applications, and therefore an efficient solution to this problem is also desired.

This thesis is concerned with improving user experience, by combining application performance with extending the battery life of mobile devices and how resources are utilized on both sides. We address the problem from the mobile and the cloud perspectives in three main parts. AlfredO uses modularization to provide optimal distributions

of rich applications between the cloud and the mobile device, and its goal is to minimize overall user interactions and reduce power consumption. The system efficiently and dynamically adjusts the application distribution to the changing conditions one is likely to encounter, such as variations in network bandwidth, user inputs or CPU load. AlfredO is effective in achieving a better utilization of mobile resources and an optimal integration of cloud-based applications with such devices. The second part proposes a performance estimation model for application deployment, that accounts for the variations in workloads and environmental factors. Based on passive measurements only, the model uses queuing networks to accurately find the optimal or nearly optimal distribution of an application in a wide range of scenarios. The final part switches the focus from the mobile device to how to efficiently distribute the cloud-based application modules in large data centers. The problem is addressed in a comprehensive and scalable manner by factoring in network, compute and availability performance aspects into one single solution.

The three parts of the thesis show that both the mobile and cloud perspectives are paramount in improving user experience in the context of integrating cloud-based applications with their mobile devices.

# Zusammenfassung

Mit der gestiegenen Beliebtheit von mobilen Geräten und dem Anstieg der Prozessorleistung sowie der Verbindung zu aktuellen Trends verlangen die Anwender immer mehr ubiquitäre und komplexe Anwendungen. Sie nutzen ihre Geräte so häufig, wie sie ihre persönlichen Arbeitsflächen nutzen, wobei sie eine ähnliche Leistungsfähigkeit erwarten, während die Akkulaufzeit erhalten bleibt oder sogar erhöht wird. Anwendungen wie Video- und Audioverarbeitung, 3D-Spiele oder Augmented Reality erweitern die Möglichkeiten von gegenwärtig verwendeten mobilen Geräten. Dabei verringert sich die Benutzerfreundlichkeit schnell, wenn mehrere dieser Anwendungen gleichzeitig ausgeführt werden. In einem entgegengesetzten Szenario, bei dem Anwendungen in der Cloud gehostet und betrieben werden, dienen mobile Geräte lediglich als Zugangspunkte. Dieser Ansatz führt jedoch häufig zu einem Anstieg von Anwendungsproblemen aufgrund von variierenden Latenzzeiten und kann nicht dynamisch an leistungsfähigere Geräte angepasst werden.

Diese Dissertation trägt dazu bei, dass solche Beschränkungen überwunden werden, indem ein flexibler Ansatz zur Integration von Cloud basierten Anwendungen bei mobilen Geräten vorgeschlagen wird. Wir setzen voraus, dass bei der Untersuchung des zwischen diesen beiden traditionellen Alternativen bestehenden Raums Teile des Rechenprozesses der Anwendungen optimal aus der Cloud auf mobile Geräte ausgelagert werden. Auf diese Weise profitieren die Anwender von der nahtlosen Integration zwischen ihren Geräten und der hohen Leistung einer Cloud. Laufende komplexe Operationen innerhalb einer Cloud reduzieren die Kosten des Mobile Computing und ermöglichen sogar Low-Entry-Modellen von der Leistungsfähigkeit einer Cloud zu profitieren. Daher machen die Anwender die Erfahrung einer Unmenge von einzigartigen Funktionen, durch welche ihre Geräte verbessert werden, wodurch sie in der Lage sind, mit einer grösseren Auswahl an leistungsstarken Anwendungen zu interagieren. Mit dem schnelleren Anstieg der Anwendungskomplexität erfordern die Cloud lokalisierten Anwendungsteile jedoch häufig multiple, vernetzte, virtuelle Vorgänge. Die Verteilung dieser Vorgänge auf die physische Infrastruktur wirkt sich auf die Art und Weise aus, wie die

Anwender mit ihren Anwendungen interagieren können. Aus diesem Grund wird eine effiziente Lösung für dieses Problem angestrebt.

Diese Dissertation befasst sich mit einer Verbesserung der Benutzerfreundlichkeit, indem die Leistung der Anwendung mit der Verlängerung der Akkulaufzeit des mobilen Geräts kombiniert wird. Sie bezieht sich zudem darauf, wie die Ressourcen auf beiden Seiten genutzt werden. Wir behandeln das Problem in drei Teilen aus der mobilen Perspektive und aus der Cloud-Perspektive. AlfredO nutzt die Modularisierung, um eine optimale Verteilung von leistungsstarken Anwendungen zwischen einer Cloud und dem mobilen Gerät bereitzustellen. Das Ziel besteht in der Minimierung aller Anwenderinteraktionen sowie in der Reduzierung des Energieverbrauchs. Das System passt die Verteilung der Anwendung effizient und dynamisch an die sich ändernden Bedingungen an, die einem wahrscheinlich begegnen, wie Änderungen der Netzwerkbandbreite, der Eingaben des Anwenders oder der CPU-Belastung. AlfredO arbeitet effektiv bei der Erreichung einer besseren Ausnutzung der mobilen Möglichkeiten sowie bei der optimalen Integration von Cloud basierten Anwendungen mit solchen Geräten. In dem zweiten Teil wird ein Leistungsschätzmodell für die Anwendungsbereitstellung vorgeschlagen, durch das die Änderungen im Bereich der Arbeitsbelastung und der Umweltfaktoren erklärt wird. Auf der Grundlage von ausschliesslich passiven Messungen werden in dem Modell Queuing-Networks genutzt, um eine optimale oder annähernd optimale Verteilung einer Anwendung in einer grossen Anzahl von Szenarien genau festzustellen. Im letzten Teil wird der Schwerpunkt von dem mobilen Gerät auf die Frage gerichtet, wie die Cloud basierten Anwendungsmodule in grossen Datenzentren zu verteilen sind. Das Problem wird in umfassender und skalierbarer Weise unter Berücksichtigung der Aspekte Netzwerk, Berechnung und Verfügbarkeitsleistung in einer einzigen Lösung bearbeitet.

Die drei Teile der Dissertation zeigen, dass sowohl die mobile Perspektive als auch die Cloud-Perspektive von grösster Wichtigkeit bei der Verbesserung der Benutzerfreundlichkeit im Zusammenhang mit der Integration von Cloud basierten Anwendungen in den mobilen Geräten sind.

# Contents


**1 Introduction** **1**
1.1 Contributions . . . . . 6
1.2 Structure of the Thesis . . . . . 7

**2 Background** **9**
2.1 Application Distribution . . . . . 9
2.1.1 Distribution Between Two-party Systems . . . . . 10
2.1.2 Performance Estimations Models . . . . . 15
2.1.3 Distribution Between Multiple-party Systems . . . . . 18
2.2 Software Modules . . . . . 23
2.2.1 Principles of Software Modularity . . . . . 23
2.2.2 Modularity in Practice . . . . . 25
2.2.2.1 OSGi . . . . . 25
2.2.2.2 R-OSGi . . . . . 27
2.3 Mobile Platforms . . . . . 29
2.3.1 Android . . . . . 29
2.3.1.1 Dalvik Virtual Machine . . . . . 30
2.3.1.2 Building Applications . . . . . 33
2.3.2 Maemo . . . . . 33

**3 AlfredO: Dynamic Software Deployment from Clouds to Mobile Devices** **37**
3.1 Motivation . . . . . 37
3.2 Challenges . . . . . 39
3.3 AlfredO . . . . . 40
3.3.1 Architecture . . . . . 40
3.3.2 Guidelines for Practical Modularity . . . . . 43
3.3.3 Code Pre-installation and Updates . . . . . 45
3.3.4 Flexible Bundle Deployment . . . . . 46
3.3.5 Instrumentation and Profiling . . . . . 48
3.3.5.1 Offline Profiling . . . . . 49
3.3.5.2 Online Profiling . . . . . 49

3.3.6 Application and Network Specification . . . . . 54
3.3.7 Consumption Graph . . . . . 55
3.3.8 Optimization Algorithms . . . . . 56
3.3.8.1 ALL . . . . . 58
3.3.8.2 K-Step . . . . . 59
3.3.9 Dynamic Adaptation of Partitions . . . . . 61
3.3.9.1 Adaptation to CPU and Network Variations . . . 61
3.3.9.2 Adaptation to Varying User inputs . . . . . 62
3.4 Summary and Discussion . . . . . 63

**4 AlfredO: Applications and Experimental Results 65**
4.1 Prototype Applications . . . . . 66
4.2 Experimental Evaluation . . . . . 70
4.2.1 Initialization Cost . . . . . 70
4.2.1.1 Nokia N810 – ThinkPad T61p . . . . . 71
4.2.1.2 HTC Desire – Amazon EC2 Instances . . . . . 72
4.2.2 Steady-State Behavior . . . . . 73
4.2.2.1 Nokia N810 – ThinkPad T61p . . . . . 73
4.2.2.2 HTC Desire – Amazon EC2 Instances . . . . . 74
4.2.3 Multiple Service Invocations . . . . . 79
4.2.4 Dynamic Optimization and Redeployment . . . . . 81
4.2.5 Reactivity to CPU and Network Variations . . . . . 82
4.2.6 Adapting to Changing User Inputs . . . . . 85
4.2.7 Resource Overhead . . . . . 86
4.2.8 Algorithms Performance . . . . . 87
4.3 Summary and Discussion . . . . . 89

**5 Performance Estimation for Mobile-Cloud Applications 91**
5.1 Motivation . . . . . 91
5.1.1 Integration in AlfredO . . . . . 93
5.2 Identifying Impacting Factors . . . . . 94
5.3 Modeling Mobile-Cloud Applications . . . . . 95
5.3.1 Application Resource Demands as Queuing Networks . . . 96
5.3.1.1 Closed, Open and Hybrid Systems . . . . . 97
5.3.1.2 Baseline Derivation . . . . . 98
5.3.2 Variations for Mobile Devices and Virtual Instances . . . . 101
5.3.3 Modeling Distribution Schemes . . . . . 105
5.4 Modeling Workload . . . . . 107
5.4.1 Model calibration . . . . . 109
5.5 Experimental Evaluation . . . . . 110
5.5.1 Model Validation . . . . . 111
5.5.2 Adaptation to Virtual Instances . . . . . 112

5.5.3 Adaptation to Device CPU Load . . . 113
5.5.4 Adaptation to Virtual Instances and Mobile Devices . . . 114
5.5.5 Comparison to AlfredO . . . 116
5.5.6 Adaptation to Workloads . . . 117
5.6 Summary and Discussion . . . 119

**6 Efficient Application Placement in Cloud Data Centers 121**
6.1 Motivation . . . 121
6.2 Problem Formulation . . . 123
6.3 Placement Goals . . . 125
6.4 Placement Algorithms . . . 126
6.4.1 Cold Spot Discovery . . . 127
6.4.1.1 Ranking of Physical Compute Nodes . . . 127
6.4.1.2 Cold Spots Generation . . . 128
6.4.2 Application Clustering . . . 129
6.4.3 Cold Spot Selection . . . 131
6.4.4 Application Placement . . . 133
6.5 Experimental Evaluation . . . 136
6.5.1 Generic Application Mix . . . 137
6.5.2 Breaking Down the Placement Technique . . . 139
6.5.3 Cache, Hadoop and Three-tiered Applications . . . 141
6.5.4 Cold Spot Discovery Threshold . . . 143
6.5.5 Comparison to Virtual Network Mapping . . . 144
6.6 Summary and Discussion . . . 145

**7 Conclusions 147**
7.1 Future Work . . . 149

**Bibliography 151**

**List of Figures 171**

**List of Tables 175**

# Chapter 1

# Introduction

Currently several trends have a significant impact on the world of computing. Until recently the resources that users had access to were limited to their local devices. Cloud computing pushes these boundaries by offering users increased processing power and storage that are easily accessible through virtualization, provided that network connectivity is available. Today, modern computing devices feature a greater ease of connectivity via LAN, WLAN or 4G networks. Due to the explosion in popularity of mobile devices, cellular wireless networks have quickly advanced from 1G to 4G, making mobile telecommunications faster and more efficient than ever. In the same time, energy efficiency is becoming increasingly relevant. For instance, cloud providers apply proactive solutions to reduce the power consumption in their data centers and mobile platforms use power management frameworks to extend the devices' battery life. To benefit from these advancements, users not only demand more complex, richer online applications and services, but also want to maintain the same fast and spontaneous interactions. In the context of these trends and demands, we need to reason about the trade-offs between energy consumption, performance provided to users and how resources, such as processing power and network, are being utilized. In most scenarios nowadays running applications on single machines is no longer a viable solution for this complex challenge, since it lacks the flexibility to find the right balance between these factors. Instead, application distribution is becoming the norm. One of the paramount scenarios that justify the motivation for distribution within the current trends is *mobile cloud computing*. This paradigm comes to alleviate the limitations of traditional approaches in providing mobile users with rich and complex applications stored in the cloud.

On the one hand, fully running such applications on mobile devices is limited by their computational resources. With the explosion in popularity of mobile devices over the last years, the latest models feature high resolution displays, multicore CPUs, dedicated graphics processors, significant storage and memory. These capabilities are increasing at an accelerated rate, thus allowing sophis-

ticated operations to be performed by pocket-sized devices. However, despite these advances, two important limitations prevent mobile devices from performing complex operations as well as their full-sized counterparts. The first limitation is battery capacity, which stops intensive tasks from running over long periods of time in order to reduce power consumption. Unlike other resources, battery density has improved only by a factor of 2 over the period spanning 1991 to 2005 (Bal06, PS05). Therefore, this limitation is unlikely to be alleviated based on advances in battery technology only. Moreover, the problem is worsened by the users' demand for lighter and smaller devices, coupled with the increasing energy demands of high resolution screens, discrete graphics and more complex CPUs. The second limitation comes from the rapid increase in users' demand for more ubiquitous applications and ever richer functionality on their devices. They want to create image albums or panoramas from photo collections, process live video, manage real-time speech recognition, and even run augmented reality or data analytics applications, while interacting spontaneously and expecting fast response times. These demands and expectations cannot be matched by what mobile devices offer today, even with the accelerated increase in their capabilities.

On the other hand, hosting and running applications in the cloud, while using the mobile device only as a thin client has its own limitations. Under a thin client model, an application is statically partitioned such that the computationally intensive parts execute remotely. The primary problem with this approach is that the application is unavailable when disconnected from the network and communication often raises usability issues due to varying latency. Furthermore, statically partitioned applications cannot adapt dynamically to newer devices that may be able to handle greater parts of the application locally. Both scenarios lack the flexibility to offer such applications to mobile users, while ensuring spontaneous interactions and crisp responses.

In this sense, the purpose of mobile cloud computing is to balance the application distribution between the mobile device and the cloud, in order to achieve faster interactions, battery savings and better resource utilization. With this, mobile devices evolve from being mere intermediaries between the cloud and the end user into true beneficiaries of cloud computing. Given the nature of cloud applications, users do not need to have the highest resource devices on the market, as complex computing operations would be run within the cloud. This lessens the cost of mobile computing to the client and allows even low-entry types of devices to take advantage of the cloud capabilities. Thus, end users will see a plethora of unique features enhancing their devices and experience. A few examples are location services, photo collections editing or others that allow users to create location based social networks. A further advantage comes from the data perspective, as mobile devices fail, may be lost or destroyed. By using the storage capability of the cloud, critical data is preserved.

Additionally, developers can benefit from mobile cloud computing. The most


**Figure 1.1:** Examples of applications: interior decoration, augmented reality, panorama generator and ticket machine (left to right, top to bottom).

important advantage of cloud computing for developers is that it allows them to have access to a larger market. Alongside the large pools of mobile applications (e.g., iTunes App Store currently holds over 500 thousand applications, while the Android Market has over 300 thousand applications), mobile users can experience richer applications that otherwise would have been out of reach or too computational intensive to allow crisp interactions. Such examples of complex applications, that given their structure are suitable candidates for distribution, are shown in Figure 1.1. For instance, in the case of the interior decoration application, the items catalog and 2D rendering can be executed on the mobile device. The 3D rendering, being more complex, is run in the cloud and periodic updates are sent to the device. Along with a bigger market, developers have the chance to build programs at a lower cost. Since programmers only need to create one version of the application and still have access to every device user, their building costs are reduced when compared with the traditional scenario in which a new application version is built for each mobile platform individually.

The idea of application distribution in the context of mobile devices and cloud computing seems to be a major step in application development and provides multi-fold benefits. It enriches users experience, expands the application market, lessens the price and effort of writing applications, reduces hardware requirements

and even provides unique chances for network operators. With this paradigm, mobile applications can take advantage of the cloud's resource capabilities and cloud-based complex applications can be integrated with mobile devices.

This thesis looks at the latter scenario, namely the integration of cloud applications with mobile devices, and is concerned with the problem of improving user experience. In our context, this means balancing the gains in applications performance with extending battery life on the mobile device and better resource utilization on both sides. We address this problem from both the mobile and the cloud perspectives. First, we start by formulating several application- and device-specific properties that need to be achieved in the client context:

- *Single usage model* – Recent studies (SHD12) show that on average iPhone and Android users have tens or hundreds of applications on their mobile devices, out of which one in four, once downloaded, is never used again. These statistics do not fit well with the idea of pre-installation, where resources are allocated for all applications, even those that are not frequently used. In addition, it makes sense to acquire on the device only those parts of an application that ensure users with lower interaction times and better resource utilization. Therefore, we argue for *on-demand migration* as alternative to pre-installation and *discarding applications after interaction* to free resources. Such a single usage model addresses the mobility aspect of the problem.

- *Short and spontaneous interactions* – Mobile users want to be able to spontaneously interact with applications and expect crisp response times, while accessing other services simultaneously. A recent survey (TKSZ12) shows that more than half of the users expect websites to load as quickly or even faster on their mobile device compared to their personal computer. Moreover, the majority of mobile users are only willing to retry an application two times or less if it is unresponsive or lagging initially, hence the need to provide them with short interactions.

- *Device and environment customization* – Diversity in mobile devices and variations in the environment factors (e.g., network latency and bandwidth, user inputs) need to be considered when deciding which applications part should reside on the mobile device. For example, consider the traditional approach in which applications run in the cloud and are accessed through a browser on the mobile device. In such scenarios, the assumption is that a browser is the only requirement to accessing cloud-based applications and that any device capable of running a browser can provide a similar user experience. In other words, the cloud application is agnostic to the mobile device accessing it. However, such devices widely differ in their features, such as screen size and keyboard, as well as in performance, security or

portability. It is clear that these attributes significantly affect the user experience. Similarly, if certain computational steps of an image processing application, for instance, are executed on the mobile device, variations in user inputs, network bandwidth or CPU load on the device greatly impact their interaction times, hence the need for customization.

- *Increased battery life* – Currently, the ability of a mobile device to consume energy outpaces by far the battery's ability to provide it. Technology trends for batteries (Pow95) show a slow increase in battery capacity when compared to processor speed improvements, thus energy remains an important bottleneck for mobile devices (PS05). Improving user experience involves prolonging the battery life, while not restricting access to rich applications.

The above properties support the improvement in user experience and specifically refer to the mobile device and the application part running on it. To address the problem from the cloud perspective, we make the transition from distribution between two heterogeneous systems (i.e., the mobile device and the cloud) to the more complex one where multiple systems (i.e., physical machines) are involved.

Virtualization-based data centers are becoming increasingly interested in efficient and scalable approaches for application distribution. In a typical IaaS, virtual instances are the basic building blocks which users can purchase to run their applications. However, most of the current solutions focus only on placing individual virtual instances within the data center. Thus, no mechanism is provided to define any additional constraints governing how these instances should be connected and deployed. For instance, the distribution should consider that a storage volume needs to be placed on a different rack than the highly computational instances. These limitations result in two important consequences. On the one hand, the problem is simplified by simply not considering structural and communication aspects of applications. On the other hand, applications are becoming increasingly complex and require multiple virtual instances with various performance and behavioral policies. Such performance policies reflect constraints on available resources, as for instance the scenario where two virtual instances must be able to communicate within a certain latency range. Behavioral policies set the expectations on how these instances respond to various stimuli in the environment. For example, a high-availability policy associated with a collection of virtual instances specifies that these instances should continue running even in the presence of failure in the infrastructure.

In this thesis we address these performance and behavioral requirements for application distribution on the cloud side and formulate several objectives that support the improvement in user experience:

- *Efficient distribution* – Consider the example of an image processing application, where one or more virtual instances performing computational

operations need to frequently retrieve images from the storage instance. If these instances are placed far apart, the communication cost increases with every hop on the routing path. Similar scenarios can occur when links on the path become congested. Thus, one needs to understand the application behavior and the resource availabilities in data centers to achieve an efficient distribution.

- *High application acceptance rate* – Cloud providers deal with thousands or more requests for application deployment daily. For instance, statistics from 2009 show that over 50 thousand virtual instances were requested in Amazon EC2's US data center over a 24-hour period. Thus, the expectation from cloud providers is to maximize the acceptance rate of application deployments.

- *Load balancing* – Effective application placements imply the distribution of requests and resource allocation in a way that minimizes the risk of overloading physical machines. Thus, load balancing improves the cloud's performance and facilitates future application deployments, by decreasing resource fragmentation.

The above properties define the desired behavior and performance of cloud providers, which impact the mobile users requesting such application deployments. Fast placement decisions and shorter response times significantly improve their perceived experience, since they are most susceptible to cloud performance degradations.

## 1.1 Contributions

The thesis addresses the integration of cloud-based applications with mobile devices, while improving user experience. This is achieved by balancing the gains in application performance, extending battery life on the mobile device and providing better resource utilization on both the device and the cloud. In detail, this thesis makes the following contributions:

**Modularity as a design principle for flexible integration and distribution of software.** The continuous growth in demand, size and complexity of modern applications and systems raises significant issues in building software, such as integration and distribution. We argue that on the one hand, modularity provides the necessary flexibility in integrating modules to build new applications. This means that developers can easily span pre-written code over different platforms. On the other hand, modularity allows the flexible distribution, even in heterogeneous environments, of modules and facilitates the performance improvement of applications.

**Optimized code and data offloading improve cloud applications performance on mobile devices.** We introduce the idea of code and data offloading as a means to enable cloud applications on mobile devices and improve their performance. An optimized offloading has the potential to significantly reduce user interactions on devices compared to traditional approaches, break through the hardware constraints of these devices and even extend their battery life. These benefits extend the range of applications supported on mobile devices with complex and rich functionality ones. We show that in order to efficiently apply offloading, optimization strategies that consider network connections, workload profiles or code and data transfers are critical.

**Efficient distribution of virtualized and networked applications in the cloud.** With demanding ever richer functionalities, applications have more sophisticated structures and claim increasing amounts of resources. Currently, a significant part of the applications hosted in cloud environments require multiple virtual instances, networked according to specific communication models. We argue that efficiently distributing such applications in the cloud can increase acceptance rate and achieve better resource allocation. Therefore, it also benefits the experience of mobile users interacting with such applications, who are more sensitive to performance degradations in the cloud.

## 1.2 Structure of the Thesis

The remainder of the thesis is structured as follows:

**Chapter 2** discusses relevant work done in the context of application distribution, with a focus on distribution between two- and multiple-party systems, as well as performance estimation models for such applications. It also introduces the concept of modularity and its principles in software engineering. It discusses aspects of physical modularity and presents software systems that manage local and remote management of modular applications. Furthermore, the chapter introduces relevant mobile platforms and discusses their architectures, as well as their programming models.

**Chapter 3** presents an outlook into our vision of integrating cloud applications with mobile devices, through dynamic software distribution. The proposed set of techniques are implemented in a prototype system, called AlfredO, built atop mobile devices and cloud virtual instances, and show how application performance can be significantly improved relative to traditional approaches. Our approach optimally distributes applications and dynamically adapts its decisions to accommodate changes in the network, data loads or the device's CPU utilization. In **Chapter 4** these ideas are validated with various applications. Additionally to

significant gains in both performance and power consumption on mobile devices, our approach shows a reasonable overhead on the mobile platform and its memory consumption is comparable to any application running nowadays on such devices.

**Chapter 5** proposes a performance estimation model to tackle the complexity of extensive profiling, required by AlfredO to optimally distribute applications. The model, integrated with AlfredO, uses queuing networks to naturally emulate application structure and behavior. Additionally, we enrich the model with performance impacting factors, such as workload, application distribution schemes, as well as resource usages of the underlying execution environment. Evaluation shows the model provides optimal or nearly optimal application distributions, while maintaining low computational costs.

**Chapter 6** looks at the more complex problem of efficiently placing virtualized and networked applications in cloud data centers. We present an approach that addresses the placement problem in a comprehensive manner, by factoring in network, compute and availability performance aspects into one single solution. Following the hardness of the problem, we introduce a resource abstraction, called cold spot, to make the problem tractable and to support increasing complexity. Simulations with a rich set of workloads show high acceptance rates, while maintaining good load balancing of resources in medium and large data centers.

**Chapter 7** concludes the thesis and places the insights gained from these different systems into the perspective of our initial vision. Furthermore, it discusses possible directions for future work.

# Chapter 2

# Background

The concepts and ideas involved in integrating cloud-based applications with mobile devices span over different areas. This chapter presents relevant work in the context of application distribution, as well as performance estimation models, and compares it with the approach presented in this thesis. Additionally, we discuss important concepts of software modularity and practical modular systems, as well as present two mobile platforms that are relevant throughout the thesis.

## 2.1 Application Distribution

There is a considerable amount of research on how to partition and distribute applications in various scenarios ranging from homogeneous environments, such as remote servers or data centers, to heterogeneous ones encompassing, for instance, mobile devices and cloud virtual instances. The problem of application distribution has become increasingly relevant, due to the ever growing complexity of current applications. At a smaller scale, this problem considers the distribution between two-party systems. Such scenarios work well for rich applications that are intended to run on mobile devices, but need to take advantage of the cloud's resource capabilities to provide better user interaction. We review relevant work in this context, from the perspective of static and dynamic approaches, in Chapter 2.1.1. An additional problem is how to efficiently estimate application distributions when the resource capabilities of the mobile devices and the cloud virtual instances used for deployment vary. From this perspective we discuss relevant performance estimation models in Chapter 2.1.2. With applications becoming more complex and many users simultaneously interacting with them, the distribution problem also needs to consider multiple-party systems. Whether the context is network embedding, topology-aware task mapping or virtual machine placement, the proposed techniques share the goal of distributing applications efficiently. We review and classify relevant work in Chapter 2.1.3.

### 2.1.1 Distribution Between Two-party Systems

One of the very early work in the context of application distribution between two-party systems was the Interconnected Processor System (ICOPS) (Sta75), based on offline resource profiling. 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 at most seven modules and only a smaller set of these could be moved between the client and server. A more recent work in this context is Coign (HS99). 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 statically partitioned to minimize execution delay due to network communication. Several other works exist such as (HF75), which applies similar ideas in the context of distributed processing over satellite-host configurations.

Our set of techniques, applied in the context of distributing applications between mobile devices and cloud virtual instances, shares with these systems the idea of building a graph model for these applications and applying graph-cutting algorithms to partition them. Similar to the previous work, our primary goal is the distribution itself and not the automatic partitioning of monolithic applications, a necessary step preceeding distribution. To meet this end, we make the assumption that applications have been already partitioned with third-party tools and the resulting components are able to communicate remotely. Such tools could be integrated with our techniques to extend the range of targeted applications. On the other hand, we note important differences to the previously mentioned systems, the first being in the concept of "distribution" itself. Our proposed algorithms do not target distributing an application on a cluster of machines or a cloud infrastructure, but rather between resource-constrained mobile devices and powerful virtual instances. In this respect, the decision is intrinsically client-driven. Second, our techniques are designed to work in dynamic and heterogeneous contexts, where clients exhibit large variability in terms of device platforms, local resources or network capabilities. Such factors need to be captured by the optimization problem and hidden to the mobile user.

**Static Application Distribution**. Static approaches, based on offline resource profiling, have been proposed in the context of "cyber foraging" (Sat01, BFS+02). Spectra (BFS+02, FPS02) and Chroma (BSPO03) partition applications into local and remoteable tasks, that are pre-installed on surrogates. Task partition is based on execution plans manually specified by programmers. While we recognize the benefit of using the programmer's knowledge to create such plans, with the ever increasing complexity of applications this approach becomes

unfeasible in practice and our techniques automate the process. Other cyber foraging systems use virtual machines techniques to provide increased flexibility. Slingshot (SF05) and Goyal and Carter's prototype (GC04) allow users to install their own functionality on surrogate machines on-demand. These systems rely on replication rather than code migration and require the programmer to explicitly deal with replication.

Research for program partitioning has been done in the context of sensor network systems like Wishbone (NTG+09), Tenet (GJP+06), VanGo (GMP+06), in the context of mobile ad hoc networks with SpatialViews (NKSI05), and in the context of cluster computing with Abacus (APGG00). Most relevant to our techniques are Wishbone and Abacus. 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 and aims at statically minimizing a combination of network bandwidth and CPU load at compile time. We recognize the benefit of minimizing the resources consumption and adhere to similar ideas when distributing applications between mobile devices and virtual instances in order to reduce user interaction time. Abacus requires programmers to explicitly partition data-intensive applications over small cluster of resources. It uses 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. On the one hand, such factors are relevant in concurrent scenarios and our techniques could borrow these ideas in the future. On the other hand, the approach proposed throughout this thesis focuses in readjusting offloading or distribution decisions at runtime, while both Wishbone and Abacus maintain static decisions during application execution.

The vision of pervasive computing (Wei91, Sat01), as the creation of physical environments saturated with a variety of computing and communication capabilities, is also relevant in the context of application distribution. 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 (WPD+02, GNM+03), or by moving around the necessary software through techniques such as mobile agents (FPV98, KBM+00). 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.

Other ongoing work on application distribution is in the context of fine-grained mobility (JLHB88), online partitioning (MGB+02) and migration of Java applications (OYZ07). A common idea is considering data objects or Java classes as the remoteable units for offloading. However, with the ever increasing complexity of current applications such approaches are not suitable for larger code bases. Other techniques have considered treating applications as three-tier struc-

tures (YGG+01, YGG+07, YSRG06). Although they benefit from putting little burden on the programmer, there is no support for dynamic migration of components or adaptation to environment factors. Odyssey (NSN+97) proposes application-aware adaptation, as a collaborative partnership between the operating system and applications, as a general and effective approach to mobile information access. The system monitors resource levels, notifies applications of relevant changes, and enforces resource allocation decisions. Then, each application independently decides how best to adapt when notified. On the one hand, we recognize the benefits of providing autonomy to applications and employ a similar idea in the context of varying user inputs. On the other hand, the distribution granularity proposed in Odyssey is much coarser, since applications can only decide whether to keep just the front-end on the mobile device and run the rest of the code from a remote server, or run the entire code on the device. This leads to sub-optimal distributions, as our evaluation shows in Chapter 4. Protium (YL01) provides support for partitioning applications, while managing replication, consistency, session management and multiple simultaneous viewers. Rover (JdLT+95) combines relocatable dynamic objects and queued remote procedure calls to manage the distribution of mobile applications. While both approaches have a coarse partitioning granularity and provide no support for adaptation to various factors, replication and consistency are important factors to be considered in concurrent scenarios and could be integrated with our techniques in the future.

**Dynamic Application Distribution**. An increasing amount of work is being done in the context of application partitioning and offloading in heterogeneous environments, especially from mobile devices to remote servers or the cloud. Most of these systems, as discussed above, tackle the static problem, that of making distribution decisions before an application interaction is initiated and without readjusting the offloading scheme at runtime. More recently, the dynamic aspect has gained attention and several approaches have emerged, although they are all based on very different premises and present different limitations. Most relevant to our work are MAUI (CBC+10) and CloneCloud (CM09, CIM+11).

We summarize the main features of MAUI (CBC+10) and CloneCloud (CM09, CIM+11) in Table 2.1 and compare them with our approach. Both MAUI and CloneCloud aim at improving the performance and battery life on the mobile device by offloading application state to either remote servers or device clones located in the cloud. Additionally, they require the application to be pre-installed at the device. While pre-installation presents benefits in reducing the remote code migration, it also creates problems with the number of platforms to be supported and as software evolves. From a cloud provider perspective, for instance, full code pre-installation becomes more expensive with an increasing number of clients. Our approach considers on-demand installation, which provides several advantages. First, it removes the need to having the software pre-installed, which

**Table 2.1:** Main features summary for MAUI, CloneCloud and our approach.

| | MAUI | CloneCloud | Our approach |
|---|---|---|---|
| Offload | mobile to cloud | mobile to cloud | cloud to mobile |
| Offload unit | method | thread | module |
| Optimizer | dynamic | static | dynamic |
| Maintains state | yes | yes | no |
| Pre-installation | yes | no | no |

is beneficial for the limited storage capacity of mobile devices. Second, it makes the problem of application evolving significantly easier. Third, eliminating this requirement reduces the overall data transfer and energy consumption on the data channel for mobile clients. In Chapter 3.3.2 we show that by considering on-the-fly code migration, our approach proves to be more network energy efficient, by a factor ranging from 4 to 10 for certain applications.

Next, we discuss the offloading units proposed by MAUI and CloneCloud and compare with our approach. Code offloading units range from fine-grained, such as methods or even classes, to coarse-grained, such as threads or functionality modules. MAUI uses methods and shows the performance improvements obtained for several applications. While this approach shows more flexibility in finding optimized distributions for smaller applications, for applications with larger code bases a coarser granularity is desired. For instance, one of our applications counts for over 900 methods heavily interconnected. By using methods as offloading units, the application distribution graph has over 900 vertices and significantly more connecting edges, which results in a very large search space. Therefore, our approach considers functionality modules as offloading units, with the goal of significantly reducing the search space and quickly making distribution decisions to adapt to environmental changes. CloneCloud operates at a similar coarse granularity (i.e., threads) as our approach. Similar to MAUI and CloneCloud, an important goal of our technique is to react to CPU load and network changes. Additionally, we consider variations in user inputs and use caching algorithm to provide applications with decision autonomy. On the other hand, CloneCloud considers only offline distribution decisions. This means that for many different user inputs and changes in external factors, such as CPU load or network bandwidth and latency, CloneCloud will not always choose the best application distribution.

CloneCloud shows the effectiveness of static analysis of Java code to manage dynamic offloading. Their evaluation shows significant gains in performance only for large inputs (i.e., 100 photos) as only then the achieved speedup on the cloud-hosted clones becomes significant. While the observed improvements are considerable, bandwidth cost is not taken into account. CloneCloud assumes that

the mobile device and respective clones are operating on fully synchronized file systems and removes the cost of such synchronization from the measurements. As soon as dynamic data is involved (e.g., computing a panorama from pictures recently taken), the observed cost in battery and performance is likely to be dominated by the data transfer. In the context of our work, we consider an alternative approach in which the cost of data transfers is 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, as shown in Chapter 4.

Both MAUI and CloneCloud maintain the application state during interaction. Although keeping application state is not the primary focus of our approach, we recognize its benefits especially for scenarios where network connectivity degrades or is lost. To this end, the techniques used in CloneCloud are compatible with our design and could be incorporated in the future.

A similar approach of using virtual machine technologies to execute the computational intensive code is discussed in (SBCD09). In the proposed architecture a mobile user exploits virtual machines to rapidly instantiate customized software on a nearby cloudlet and to use it over WiFi. A cloudlet is a trusted, resource-rich computer or a cluster of computers well connected to the network and available for use by nearby mobile devices. Rather than relying on a distant cloud, the cloudlets eliminate the long latency introduced by wide-area networks for accessing the cloud resources. As a result, the responsiveness and interactivity on the device are increased by low latency, one-hop wireless access to the cloudlet. On the one hand, these ideas are practical and have the potential of improving even further the performance obtained with our approach. On the other hand, the mobile device acts as a thin client, with all the significant computation occurring in a nearby cloudlet. Additional to a lack of flexibility in code distribution, this approach relies on a technique called dynamic virtual machine synthesis, that as reported by the authors requires 60 to 90 seconds at bootstrap. Therefore, it might not be acceptable for enabling ad hoc user interactions.

Zhang et al. (ZJKG10, ZSG+09) develop a reference framework for partitioning a single application into elastic components with dynamic configuration for execution. The components, called weblets, are autonomous functional software entities that run on the device or in the cloud and perform computing, storing and network tasks. An elasticity manager component transparently decides on the instantiation and migration of weblets, but only considers the network bandwidth and latency as relevant factors.

Odessa (RSM+11) 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, Kwon and Tilevich (KT12) have 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 can therefore not be ignored. When referring to remote execution for application parts, most of the existing dynamic approaches offload the application state only, while ignoring the hurdles of both code and data migration. We argue these problems are essential in realistic scenarios and need to be addressed, thus our techniques provide support to offload code and data on-the-fly.

### 2.1.2 Performance Estimations Models

Given the multitude of mobile devices, cloud virtual instances or application user inputs, the application behavior changes whenever such environmental variations occur, a situation bound to frequently occur in practice. In such scenarios, dynamically distributing applications between mobile devices and cloud virtual instances based solely on statistics gathered for static environments is no longer possible. Moreover, optimal decisions require extensive application monitoring for each distinct combination of varying factors. To reduce the computational complexity of such an approach, we consider performance estimation models that are able to take optimal or nearly optimal decisions based on passive measurements. First, we review theoretical models based on queue networks. Second, we discuss different applications of such models to practical performance estimation problems and focus on two relevant works (UPS+05, UPS+07) and (SKZ07). Finally, we summarize several workload characterization studies that have identified regularities in modern application workloads.

**Queuing Networks Based Models**. Queue networks have been widely studied and Jain (Jai91) describes practical applications of queuing theory to computer system performance analysis. Most of the described queue network models assume relatively detailed information about transaction or interaction behavior, such as resource utilizations, service and waiting times, as well as visit ratios. This information is easily accessible through measurements at application and infrastructure-level, thus we adhere to the idea of building a performance estimation model based on queuing networks.

Operational analysis in queuing theory attempts to avoid probabilistic assumptions about system workload, such as Poisson arrivals and exponentially-distributed service times, and relies only on measured quantities (DB78). Little's law and the Utilization law are classic examples of operational laws. Mean Value Analysis (MVA) focuses on the averages of performance measures, as opposed to the full distributions. This method applies to closed queuing networks and fits well with our class of applications. Generalizations of queue networks address multiclass networks (Cas06). Workload classes are often interpreted as categories such as "batch", "terminal", and "transaction", but classes can also represent

different transaction types. In principle such multiclass networks could potentially work well in our context, since we extract sufficient information about user interactions to employ them. However, the computational cost of extracting exact solutions increases rapidly with the number of transaction types. To avoid this, for modeling workload we looked at nonstationarity of interactions as the basis for performance estimation.

Previous work has proposed complex queuing models capable of capturing simultaneous resource demands and parallel subpaths that occur at the tier-level of single or multitier applications. A G/G/1 model for replicated single-tier applications, such as clustered Web services, is proposed in (USR02). Similar work (LNP$^+$03) uses M/M/1 queues to compute response times of Web requests. For multitier applications, approaches such as (RS95, WR95, LKL01, XOWM06, Fra99), represent the fact that software servers are executed on top of other layers of servers and processors. Mostly, the focus in these papers is on application tiers developed for homogeneous environments, whereas our work is applied in the context of applications composed of multiple modules and distributed between mobile devices and cloud virtual instances. More similar efforts are reported by Kounev et. al (KB03) and Bennani et. al (BM05). Kounev and Buchmann propose a queue network-based model for the performance prediction of a 2-tier SPECjAppServer2002 application and solve this model numerically using open source analysis software. Bennani and Menasce model a multitier Internet service serving multiple types of transactions as a network of queues with customers belonging to several classes. The authors employ an approximate MVA algorithm to develop an online provisioning technique using this model. We consider a similar approach based on MVA and apply it on estimating the response times of application interactions in order to optimally distribute it in the mobile – cloud setup. A further difference to their work is that we validate the model using realistic applications, while their evaluation is based on simulations with online and batch workloads.

**Performance Prediction**. Most relevant to our work are two techniques that apply queuing models to distributed applications. First, Urgaonkar et. al (UPS$^+$05, UPS$^+$07) model multi-tier Internet services with queue networks and exploit MVA to compute average response times. In some respects this work is similar to that of Liu et. al (LHS05) and the model has certain similarities and differences from our approach. Both models associate a single queue to each tier or application module, respectively. In addition, our approach requires similar parameter estimations, including visit ratios at each tier or module, service times and waiting times. An important difference is given by the assumptions about how requests circulate among tiers or modules. In the model proposed by Urgaonkar et. al the application assumes a pipelined structure, which means that any application tier, except for the entry tier, can receive requests only from the single previous tier. In contrast, in our scenarios the applications are rep-

resented by directed acyclic graph structures, which means that an application model can receive requests from an arbitrary number $N$ of dependent modules. Another relevant difference is related to the types of applications used for evaluation. While their model reports accurate average response time estimations for two sample applications (RUBiS and Rubbos) subjected to stationary synthetic workloads in a testbed environment, our model is evaluated on real applications, that additionally exhibit nonstationary mixes of interactions.

Similar to (UPS$^+$05, UPS$^+$07), Stewart et. al (SS05) considers a performance model for distributed Internet applications based on using profiles that summarize information about how application components and related workloads place demands on the underlying system resources. On the one hand, we borrow some features in our model, namely accounting for inter-component communication and component placement, as well as considering waiting times by an M/G/1 model. On the other hand, while their approach describes the workload by a constant scalar arrival rate, our model uses a time-variable vector corresponding to counts of interaction types. A second difference regards the workload used, which in (SS05) are stationary and synthetic, unlike the nonstationary mixes of interactions we consider. Furthermore, their method requires expensive calibration, since the resource consumption profile for each component must be estimated via controlled benchmarks on dedicated machines per components.

Finally, a more recent work from Stewart et. al (SKZ07) focuses on the nonstationarity characteristic of transaction mixes over time. We find important similarities between transactions and interaction mixes and, therefore, borrow several aspects of their approach to model workloads in our scenarios. First, their assumptions on the request-reply nature of transactions and the limited number of transaction types encountered in Internet applications also apply for our class of applications. Second, their basic, extended and composite models potentially fit with our performance problem, since we provide all required estimations on service and waiting times, as well as resource utilizations. However, we note several important differences as well. First, their approach explicitly models distinct physical resources such as CPU or disk by associating dedicated queues, while our model provides one queue per application module. Second, the calibration in our approach is based on characterizing the behavior of application functionality modules and communication links as function dependent on the inputed workloads. In (SKZ07) the calibration phase is focused on identifying parameter values to be associated to transaction types. Finally, although both models evaluate the accuracy of the models on real applications, our work is targeting distributed mobile-cloud applications, while their approach is exclusively directed at Internet applications.

**Workload Characterization and Modeling**. Previous work (AW96) has characterized important workload aspects in modern transactional applications. Menasce et. al (MAFM99) propose first-order Markov models of customer ac-

cesses at e-commerce sites. Cao et. al (CCLS01) has studied the nonstationarity characteristics in workloads, such as Internet traffic or user request cycles to Web sites, without considering mixes of application-level transaction types. Other approaches consider clustering techniques to simplify workloads for performance modeling, such as according to their resource demands as in Magpie (BDIM04). To understand these resource demands for different transaction types, the authors exploit knowledge of application architecture. Rolia et al. (RKKD10) proposes a resource demand modeling approach, while others attempt to diagnose performance changes by comparing request flows from two executions (SZdR$^+$11), or to estimate performance for embedded systems based on discrete event simulations (MDA07). Compared to previous work, Stewart et al. (SKZ07) looks into the nonstationarity of application transaction types, based on the observation that these types cluster workload elements according to their resource demands. As a result, they show that nonstationarity in transaction mixes allows calibrating performance models without invasive instrumentation or controlled benchmarking, while relying solely on lightweight passive measurements. Although this approach looks at Web transaction-types of workloads, it is relevant to characterize and model workloads for mobile-cloud applications as well, where the types of interactions users can perform per application are limited to several categories (i.e., log in, image processing or search in a indoor localization application). Our estimation model borrows the nonstationarity idea to deal with workloads and adjusts it to match the heterogeneous context we consider.

Some work (TDZN10) has also been done in modeling the performance for complex and popular applications, such as Mozilla, Visual Studio or Microsoft Office. The proposed model shares from both black-box and white-box models, by using the developer's knowledge to understand what parts of an application to instrument and statistical algorithms to answer what-if questions. The applications consider concurrent states from multiple users and in order for the model to answer performance questions relevant attributes are filtered based on similarity search. In (WMG$^+$10), the authors look into performance modeling of virtualized resource allocation, based on probabilistic relationships between virtualized CPU allocation and application response time. In (SPPM11), the authors introduce a method based on linear regression and clustering to predict performance requirements of mobile devices tasks using hardware resource utilizations and input data. However, to the best of our knowledge none of the existing models address the problem of estimating the performance of applications distributed between mobile devices and cloud virtual instances.

### 2.1.3 Distribution Between Multiple-party Systems

Distributing applications between multiple-party systems represents a more complex problem than considering only two-party systems. The ever increasing func-

tionalities of recent applications makes this topic increasingly important, since placing them in a cloud environment requires several virtual instances per application with specific demands and constraints, interconnected by a communication model. We discuss the placement problem in different contexts and present a comparative analysis of relevant techniques while noting relevant opportunities and challenges.

**Virtual Network Placement and Provisioning**. The application placement problem shares similarities with the virtual network mapping problem, which plays a central role in building virtual networks. During the mapping process each virtual node is assigned to a physical node and each virtual link is mapped to a path or flow in the physical network, such that resource constraints are satisfied. In general, the virtual network mapping problem seeks at making efficient use of the underlying resources in a physical network, while still satisfying the existing mapping constraints. This problem has practical implications for virtual private network provisioning and virtual network embedding. One main difference is that in mapping virtual networks, physical nodes have dual roles, of both satisfying computing demands and routing traffic. However, this assumption does not hold in data center networks and only considers computing and network resources. In mapping virtualized applications, we are additionally concerned with constraints such as locality and availability.

Several heuristics have been developed to solve the virtualized application mapping problem (YYRC08, ZSB$^+$08, RAL03, SIB03). Some restrict the search space by only solving the link embedding problem and assume that the node placement is given in advance (SIB03). In (RAL03) the authors propose a simulated annealing approach to map virtual networks on Emulab (Emu11). To simplify the node placement problem, all virtual nodes are associated with a type and nodes mapped to the same physical node must have the same type. Notably, a divide-and-conquer approach is presented in (ZA06), where a collection of connected physical resources is first identified as the target for placement, instead of the entire system. A follow-up on this technique is presented in (ZSB$^+$08), in which authors decompose the network substrate and the virtual networks into well connected clusters and star topologies, respectively. An important step of our technique, that of identifying highly available collections of physical machines, is inspired from this work, but extended to efficiently scale to large environments. Oki et. al (OI09) compares performances of optimal routing by the pipe, hose, and intermediate models in the context of communicating virtual machines.

In (YYRC08) the authors advocate that to solve the network embedding problem it is crucial to extend the network substrate to support traffic splitting and link migration. Although our approach does not require additional network support, we believe such network capabilities may be useful to further improve our technique at run time. In (LT06) the authors tackle the problem of mapping with a backbone-star topology to the physical network, such that a subset

of nodes constitute the backbone and the remaining nodes each connect to the nearest backbone node. Razzaq et. al (RR10) and Chowdhury et. al (CRB09) both propose a two stage solution, the difference between them being that the latter assumes substrate support. Butt et. al (BCB10) notes the relevance of differentiating between physical network resources and argue that topology-aware mapping and reoptimizing bottleneck mappings have the potential of improving the acceptance ratio. To discover available sets of physical machines, Macropol et. al (MS10) have proposed a technique to probabilistically search top scoring clusters on large graphs in linear time. Their evaluation on real network graphs from Amazon or Wikipedia show that the algorithm is more efficient than the state-of-the-art MLR-MCL (SP09) and MCL (vD00). An alternative approach to clustering is based on stochastic flow injection and proposed in (YCL95). The goal of minimizing the clustering cost is transformed into a uniform multicommodity flow problem by adding artificial weight functions, solved in a probabilistic manner to reduce complexity. These techniques could be tailored to enhance our technique.

**Topology-aware task mapping**. Due to the increasing size of parallel computers, the interconnection network has become the system bottleneck for applications and processes running on these machines. As a result, placing task graphs on a network of processors has gained much attention. This problem is similar to ours, in that task nodes must be placed on processor nodes, while satisfying the resource constraints of both network and processors. The key differences are the following: (a) resource constraints are specified coarsely (e.g., communication demand between tasks is quantified as "high" and "low") and contrasts our problem, in which users impose fine-grained resource requirements on the cloud provider to achieve the desired application performance; (b) most solutions ignore compute resource constraints and focus on maximizing communication throughput only; (c) no availability constraints are considered.

Typically, task-mapping algorithms consist of two stages, namely partitioning and mapping. In the partitioning stage, tasks nodes are clustered based on their communication demands. In the mapping stage, individual task nodes are placed such that tasks within the same cluster are placed on the same processor whenever possible. We also recognize the importance of clustering virtual nodes to enable efficient heuristics and have adopted a novel clustering technique in our work. In (TC00), the authors propose a graph-matching inspired algorithm that seeks at maximizing throughput. The technique is based on finding subgraph isomorphism and it is highly combinatorial. More recently, in (ASK06) the authors present a topology-aware placement technique that takes into account the topology of the processor network when making placement decisions. In (KRTP00) the authors look at mapping clustered task graphs onto multiprocessor architectures. Their heuristic finds a mapping by first placing the highly communicative tasks on adjacent nodes of the processor network. The remaining tasks are placed

by starting from those close to the already placed ones. In (EW97) the authors propose a sub-optimal, but efficient, algorithm for task mapping based on multilayer clustered graphs. Srinivasan et. al (SJ99) also focuses on task mapping in distributed systems, but with a focus of maximizing reliability. In (BSSH05), the operating system kOS is presented, as a biologically-inspired approach to support the operation of task mapping and scheduling in sensor networks. Although all previous approaches could be applied to our problem, neither considers fine-grained resource constraints.

**Network-aware virtual machine placement in cloud environments**. An important work that looks at virtual machine placement in clouds is (MPZ10). The authors formulate the virtual machine placement problem as an optimization problem and prove its hardness. However, the proposed heuristic is limiting in that it assumes a physical machine can only host one virtual machine at a time. More recently, Jayasinghe et. al (JPE+11) presents a heuristic algorithm based on clustering that is shown to work in an empty system. In (SZL+11) the authors consider the problem of dynamic placement only, in which they seek to mitigate resource bottleneck by migrating virtual machines between physical machines. Although advancing interesting ideas, these solutions cannot scale, since the algorithms can potentially consider all physical machines in the cloud for making one placement or migration decision. By identifying available sets of physical machines in advance we can significantly reduce the search space for dynamic placement. In (BASS11) a novel cloud network platform called CloudNaaS is proposed which extends the self-service provisioning model of the cloud beyond virtual servers and storage to include a rich set of network services. Relevant to our work is that in CloudNaas the placement of virtual machines is fully decoupled from the placement of network demand. This approach leads to resource fragmentation, since compute and network resource can be unevenly utilized. We advocate for a more coupled approach that considers the management of both network and compute resource into one single integrated solution.

Previous work on virtual machine placement mainly focuses on optimizing the efficiency and effectiveness of computation resources utilization, allowing virtual machines to share the capacity of these resources on the physical hosts. The placement problem is modeled as either a constraint satisfaction problem (VTM09) or a constraint multiple knapsack problem (TSSP07) to maximize the utility of computing resources. Other approaches (PY10) focus on minimizing the data transfer time consumption, by placing virtual machines as close as possible to the required data. However, none of these solutions consider both the computing and network resources into one solution. With the first approach, virtual machines would be allocated within a non-optimized physical distance to the relevant data, which would cause an extra data transfer overhead and eventually lead to the degradation of application completion time. With the second approach, virtual machines could be allocated on already highly utilized physical hosts, which would

slow down their execution time.

Further, the cross entropy technique (RK04, Rub99, HBKK05, CSLZ10) has also been considered in (SD05), to tackle the placement of parallel applications. The difficulty here is that one has to analyze a large number of samples, hence hindering the possibility of dealing with a large system. To overcome this limitation, Tantawi (Tan12a) proposes a statistical enhancement over the classic technique that biases the sampling process to incorporate communication needs and availability constraints. Results show, however, a higher network fragmentation in the data center.

**Graph matching**. Finally, graph matching has been a classic approach to tackle the problems of virtual network provisioning, virtual machine placement or task mapping. In its simplest form, one wants to construct the isomorphism between two given graphs, where nodes have no labels. Many efforts have been spent to devise such isomorphism algorithms (Ull76, SD76) that are feasible under practical conditions. A common approach is that of looking for an optimal representation of the search space and choosing effective pruning policies. The VF algorithm (CFSV99) is based on a depth-first search strategy, with a set of rules to prune the search tree, and compared to (Ull76) proves to be more efficient (i.e., the memory requirement is $O(N^2)$ instead of $O(N^3)$).

An improved VF algorithm, called VF2, for large graphs has been proposed in (CFSV01) and is based on backtracking. Its complexity is significantly lower than previous algorithms, and in certain scenarios it can even be executed in linear time. The VF2 algorithm has been applied to solve the virtual network provisioning problem in (LK09), to map nodes and links during the same stage. The advantage of using graph isomorphism is given by the fact that when a non-valid mapping decision is detected, it can be revised by simply backtracking to the last valid decision. In this scenario, the traditional approaches, where the links and nodes mapping is separated in two stages, have to remap all links and become very expensive in terms of runtime. Recently, an improved version of VF2, called VF2x, has been proposed to solve the virtual network mapping for testbed workloads (YR12). Similar to (LK09), we have also recognized the inefficiency of traditional approaches and consider compute and network resources, as well as availability constraints, into one integrated solution.

Graph matching techniques have also been used in the context of task mapping (ST85). In these scenarios, graphs represent the module relationship of a given task and the processor structure of the distributed system. Thus, module assignment to processors becomes a type of graph matching, namely weak homomorphism. The effectiveness of this approach is improved by solving the homomorphism problem through A* search. Additionally, pattern recognition benefits from graph matching as shown in (GMS05, CFSV04). In (CFSV04), the authors improve the VF2 algorithm by reducing its space complexity and evaluate its efficiency on graphs with up to 1000 nodes. An alternative approach

for exact and approximate graph matching using Markov random walks was proposed in (GMS05). The authors show their method is more efficient than the VF algorithm, especially in scenarios of image retrieval tasks. While graph matching would be applicable for virtual application placement, the problem becomes more difficult when availability constraints are involved. The graph isomorphism algorithm should be modified in order to consider availability constraints, thus we believe such an approach is not well suited in our context.

## 2.2 Software Modules

Applications are becoming more complex and oftentimes they are assembled from pieces of code developed independently. Modularity involves the breakdown of these systems and applications into separate components, called *modules*, with the purpose of improving their flexibility and comprehensibility, as well as dealing with the increased complexity. A module represents an encapsulation unit with a specific functionality or closely-related functionalities, and consists of data, functions and objects that are specific to the programming language used. The encapsulation property, also referred to as *information hiding*, means that the content of a module is not entirely exposed to other modules or applications. Instead, only its interfaces are visible and any information about a module, like its concrete implementation, is hidden unless specifically declared public.

### 2.2.1 Principles of Software Modularity

Applying modularity to design systems represents a two-phase process. The first stage consists of the problem decomposition, namely a top-down approach that divides a problem into smaller and less complex subproblems. In (Par72), it is shown that the effectiveness of modularization depends on the criteria used in dividing a system into modules. On the one hand, modules can be easily generated through conventional ways, for instance based on flowcharts. However, such an approach usually means all modules are affected when changes occur in the system. On the other hand, one may use the concepts of *information hiding* and *separation of concerns* (Dij74), to generate modules that are independent of each other, such that changes are ideally confined within a single module or propagate to a small number of modules. This means that the correctness of a module should not depend on other modules, and thus the effects of faults should be restricted to the module where the fault occurs. Due to module dependencies that enable sharing between them, the propagation of faults can occur in practical applications and systems. However, even in such scenarios the desire is that only dependent modules are affected and not parts of the system that are unrelated. This means that any communication between modules is managed through the

explicit interfaces exposed and there is no hidden sharing between them.

The second stage in building modular systems is composition, as a bottom-up approach where modules are combined to result in new systems. This means that designing software modules should ensure their reuse in systems whose purposes are quite different from the one in which they were initially developed. In practice, limiting the number of dependencies of modules is an important step to achieve good compositions. By studying the behavior of a system, we can understand how changes are propagated to all modules. For instance, modules with few incoming dependencies are easier to change than those with many such dependencies. Likewise, software modules with few outgoing dependencies are easier to reuse than those with many outgoing dependencies. Therefore, the balance between reuse and maintainability is an important factor to consider when designing software modules and dependencies play an important role. In general, a good practice in modularization is for modules to share information through as few dependencies as possible. These ideas that are related to the separation of concerns have been expressed as a set of guidelines for evaluating modularity in (Mey88).

Module cohesion (SMC74) also needs to be considered when designing high quality software modules and represents the extent to which the functionality implemented by a module is related. Generally modules should be highly cohesive, ideally exposing only a single functionality. A module with too little behavior is not particularly useful to other modules, and therefore has minimal value. On the contrary, a module implementing too many unrelated functionalities is difficult to reuse, because it provides more behavior than desired. When designing software modules, identifying the right level of functional granularity is an important and difficult task.

Another measure of the quality of software modules is coupling. Whereas cohesion is used to describe a single software module, coupling refers to the relationship between modules. As such, it determines the extent to which one module depends on other modules. One can define flavors of coupling ranging from high to low. An example of high coupling is when a module modifies or relies on the internal workings of another module, as for instance accessing its local data. On the contrary, low coupling assumes that the communication between modules is managed though message passing or indirect invocation methods, such as events. In modularity, high coupling should be avoided, since it defeats the purpose of modules reuse, affects their independence (i.e. the behavior of the dependent module is directly impacted by the changes in the implementation of the other module) and makes it harder to understand them in isolation. There are scenarios where two highly coupled modules implement different parts of the same functionality, in which case it is better to group their implementations under the same module. Looking at coupling, structural dependencies represent the means to measure direct data communication between modules. However, two such modules con-

nected to the same set of other modules might indicate a similar purpose or functionality, a property referred to as fan-out similarity. Evolutionary coupling reflects implicit dependencies between modules, driven by scenarios where these are frequently changed together during development. Current tools oftentimes consider coupling as a one-dimensional concept, by equating coupling with structural dependencies. Exploiting combinations of coupling concepts, where fan-out similarity or evolutionary coupling are additionally considered, could improve such tools and provide better guidelines for applying modularity in practice.

### 2.2.2 Modularity in Practice

Modularity can be classified as *logical* or *physical*. Logical modularity refers to the internal organization of code into logically-related components. In modern languages such as Haskell or Modula-2, logical modularity starts at the class level, the smallest code unit that can be defined, and can progress at the level of packages, which organize code into groups of related classes. Regardless of the implementation scale of such modules, functionality should be exposed through simple interfaces that shield callers from further code details.

Physical modularity refers to the actual software creation and consists of two main components: first, one or more compiled modules, and second, an environment that understands how to run these modules. Therefore, unlike logical modularity which requires that a specific composition of modules is passed to the compiler and linker to generate a single binary, physical modularity preserves the module boundaries. Representative implementations of physical modularity are shared libraries, Java JAR files or the Component Object Model (COM) and OSGi standards. In this thesis, we are concerned with the physical modularity aspect and more specifically with the OSGi (OSG07) standard, discussed in the next section, as well as R-OSGi (Rel11), an important extension that achieves the distribution of modular applications over remote machines.

#### 2.2.2.1 OSGi

OSGi (OSG07) provides a general-purpose, secure and managed Java framework that supports the deployment of extensible software modules, called *bundles*. A bundle is comprised of Java classes and other resources, which together provide functions to end users. Bundles can share Java packages among an *exporter* bundle and *importer* bundle in a well-defined fashion. In OSGi, bundles are the only entities used for deploying Java applications. Bundles are packaged as JAR files that store applications and their corresponding resources. We describe these resources as follows

- resources necessary to provide various functionalities – these resources may be Java class files, as well as other data such as HTML files or icons. A

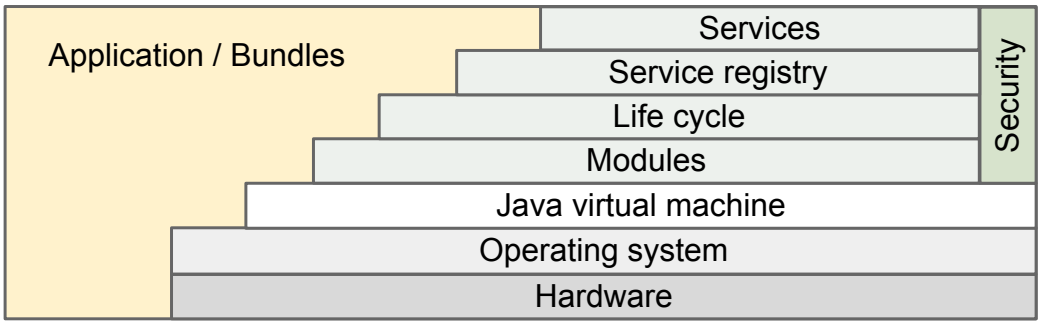



**Figure 2.1:** OSGi architectural overview.

JAR file can also embed additional JAR files that are available as resources and classes.

- `Manifest` file describing the contents of the packaged bundle and providing additional information – this file uses headers to specify information required for the framework to correctly install and activate the bundle. For instance, it states dependencies on external Java packages that must be made available to the bundle before it can run.
- optional documentation that can store the bundle's source code. However, management systems may remove this information to save storage space.

OSGi-compliant devices can download and install bundles, as well as remove them when they are no longer required. The framework manages the installation and update of bundles in a dynamic and scalable fashion. Thus, it provides the bundle developer with the resources necessary to benefit from Java's platform independence and dynamic code loading capability in order to easily develop applications for memory constrained devices.

The functionality provided in the OSGi framework is divided in several layers, as shown in Figure 2.1. The *security layer* is based on the security provided by standard Java, but adds a number of constraints. It defines a secure packaging format and the runtime interaction with the Java security layer. The *module layer* defines a modularization model for Java and addresses several shortcomings of the Java deployment model. The modularization layer provides strict rules for sharing Java packages between bundles or hiding packages from other bundles. The *life cycle layer* provides an application programming interface to manage the bundles from the module layer. It defines how bundles are started and stopped, as well as how bundles are installed, updated and uninstalled. Additionally, it provides a comprehensive event model to allow a management bundle to control the operations of the service platform.

The *service layer* provides a dynamic, concise and consistent programming model for developers, simplifying the development and deployment of bundles by

decoupling the service specification (i.e. interface in Java) from its implementation. This model allows bundle developers to bind to various services only using their interface specifications. The selection of a specific implementation through the service registry, optimized for certain needs, can thus be deferred at runtime. The programming model provides consistent interfaces, which support developers to cope with scalability issues, such that software components can be mixed or matched and still result in stable systems. Bundles register new services, receive notifications about the state of services, or look up existing services to adapt to the current capabilities of the device currently in use. This aspect of OSGi makes an installed bundle extensible after deployment, such that new bundles can be installed to add features or existing bundles can be modified and updated without requiring to restart the framework.

*Versioning* is one of the most important features of OSGi, such that all packages or bundles that can evolve over time are versioned. The problem addressed with versioning is the independent evolution of dependent artifacts. In OSGi terms, a bundle can import a package exported by another bundle, and thus develops a dependency on the exporter. An importer uses a specific exporter during the compilation and build process and ideally, the same exporter is used at runtime as well. This fidelity reduces the number of potential problems because many aspects are guaranteed to be verified during the build process. To achieve this, version ranges are used and embed compatibility policies specified by the importer (i.e., major policy means an incompatible update, minor policy signals a backward compatible update and micro policy reflects a change not affecting the service level).

#### 2.2.2.2 R-OSGi

Since OSGi is applicable for centralized applications only, R-OSGi (Rel11) allows their transparent distribution, without any changes to their implementation or structure. To provide remote communication between application bundles, R-OSGi generates dynamic proxies at the service boundaries. R-OSGi achieves transparency with the following techniques: (a) dynamic proxy generation, (b) distributed service registries, (c) type injection to resolve distributed type system dependencies, and (d) mapping network and remote failures to module events. The architectural overview of R-OSGi is presented in Figure 2.2.

**Service proxies.** R-OSGi ensures distributed communication by dynamically creating proxies for remote services, such that a client cannot distinguish between a local and a remote service. Proxies redirect all service method calls to the remote service and propagates the result to the local service client. Such dynamic proxies allow spontaneous interactions between services and reduce the amount of data that is transferred over the network when a client binds or fetches services. To create a proxy for a service interface requires the bytecode analysis of

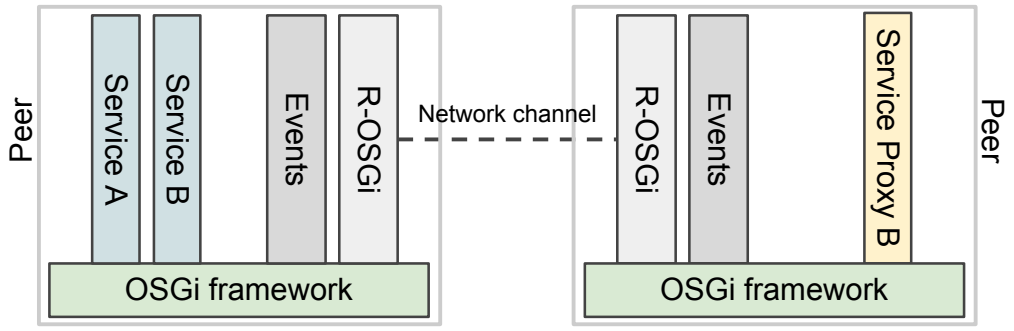



**Figure 2.2:** R-OSGi architectural overview (adapted from (Rel11)).

the original service at registration time. At fetching of the service interface, the provider offers the corresponding Java bytecode for the interface, that is required to generate the full proxy. The operation of creating dynamic proxies is efficient, especially for resource-constrained devices, since no method definitions or other code need to be transferred to the client.

**Service registry.** In order to distribute monolithic OSGi applications, the centralized service registry provided by the OSGi framework does not provide the proper functionality. This means that a distributed service registry is required and there are two approaches to provide it. On the one hand, the OSGi framework can be modified to provide support for such a registry. However, this solution can hinder the generality of the OSGi platform. On the other hand, external protocols can be used, such that proxies can be registered as local services in the conventional registry. R-OSGi takes this second non-invasive approach.

OSGi uses an explicit binding model in which each service announces its availability in a proactive fashion. However, if we consider a large distributed system or application, with a large number of bundles and services, such an approach may suffer from scalability issues. To avoid this problem, R-OSGi changes the service discovery to being reactive, which means that services are discovered only when entities in the system announce their demands for those services. If a new service is registered with the framework and its properties show that it should be remotely provided, R-OSGi registers the service with the discovery layer.

**Type injection.** In OSGi, all applications are modularized and code imports within a bundle from other bundles need to be explicitly declared in its manifest file. When looking at proxies, certain implications need to be considered. First, the service interface might use types in parameters or return values that are not Java standard and, thus, cannot be assumed to be present on the client side. In order to ensure that proxies are resolvable, R-OSGi uses type injection to make these proxies self-contained and to ensure type consistency. When registering the remotely accessible service, static code analysis determines all the types used in

the service interface. If a specific type is already contained in the bundle, its corresponding class is added to the injection list. When a client fetches the service, the injections list is transmitted along with the interface and corresponding properties. At proxy generation, these injections are stored in the proxy and declared as exports to ensure type consistency. As a result, the injection generates a minimal set of classes and package imports to make the proxy resolvable.

In distributed systems, one has to consider the potential problems of communication latency or unreliability, that may lead to failures. Instead of hiding these failures, R-OSGi exposes such events to application bundles, which handle them by module unloading. For example, if a remote service provider fails, the platform detects the network channel breakdown and the impossibility to reconnect, which triggers the uninstallation of the proxy.

To conclude, R-OSGi provides support to build distributed applications in a transparent fashion, using the same modularity features and capabilities of the OSGi platform. Additionally, R-OSGi maps failures of distributed applications onto unloading events that bundles are designed to handle. Experiments with different benchmarks show that R-OSGi has a similar or better performance than alternative distribution mechanisms, such as RMI (Remote Method Invocation) or UPnP (Universal Plug and Play) (UPn00).

## 2.3 Mobile Platforms

Recent developments in mobile computing have been outstanding, resulting in devices that feature advanced computing abilities, media-related and connectivity options. The most popular mobile operating systems used today by smartphones and other hand-held devices are Google's Android (And07), Apple's iOS (IOS12), Microsoft's Windows Mobile (Win12) and Nokia's Maemo (Mae05). In the remainder of this chapter, we discuss the Android and Maemo mobile platforms, which are relevant throughout the thesis.

### 2.3.1 Android

Android (And07) is Google's Linux-based operating system for mobile devices and is usually referred to as a software stack. Each layer of the stack groups together several programs that support operating system functions, as shown in Figure 2.3, such that the stack base represents the *kernel*. Google uses the Linux version 2.6 to build the Android kernel, which includes memory management programs, security settings, power management software and several hardware drivers.

The next level includes the Android *libraries*. For example, the media framework library supports playback and recording of various audio, picture and video

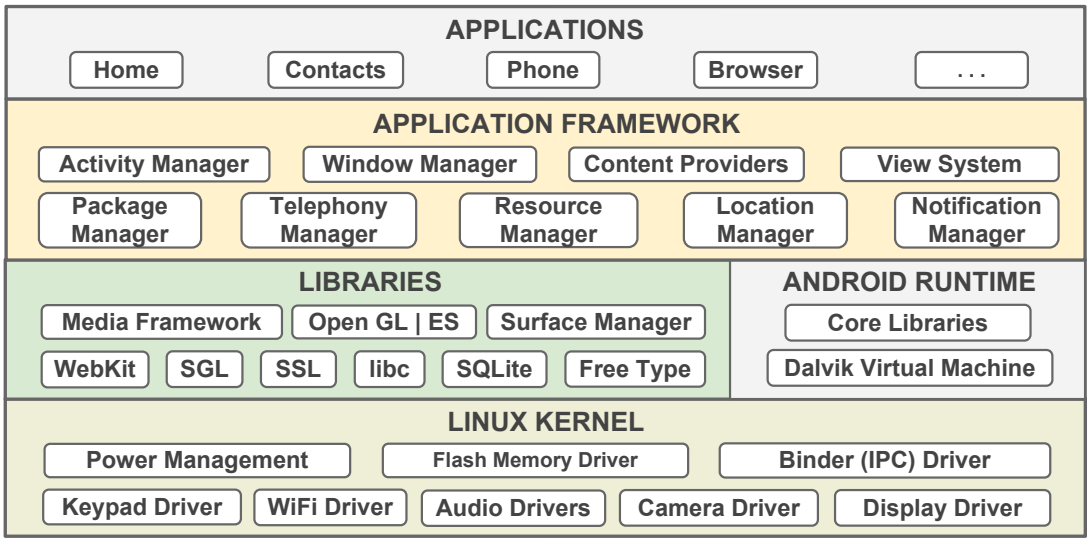



**Figure 2.3:** Android architectural overview.

formats. Other libraries provide three-dimensional acceleration or Web browser support. Adjacent to the libraries layer, the runtime layer includes a set of core Java libraries, used to build applications, and the Dalvik virtual machine. Running each application in its own virtual instance has several benefits: (a) applications are independent of each other, (b) application crashes should not affect other running applications, and (c) memory management is simplified. We provide more insight on the Dalvik design in Chapter 2.3.1.1.

The next layer represents the *application framework*, which includes the programs that manage basic functions like resource allocation, switching between processes or keeping track of the device's physical location. Application developers have full access to the application framework and can, therefore, take advantage of the offered processing capabilities and support features when building an Android application. The top of the stack contains the *applications* and the basic functions of the mobile device, such as making phone calls, accessing the browser or the contact list.

#### 2.3.1.1 Dalvik Virtual Machine

The Dalvik virtual machine was developed for two main reasons. First, as part of the Android application runtime, it can provide support for a diverse set of mobile devices. Second, it allows applications to be sandboxed for security, performance and reliability. However, a virtual machine-based runtime does not necessarily balance these requirements with the limited processor speed and limited RAM that characterizes most devices. To address these conflicting requirements, every

Android application runs in its own process, with its own instance of the virtual machine. To support multiple applications, Dalvik has been written such that a device can efficiently run multiple virtual machines. Additionally, Dalvik executes files in the `.dex` format which is optimized for minimal memory footprint. The virtual machine is register-based and runs classes compiled by a Java language compiler that have been transformed into the `.dex` format by the included `dx` tool. It also relies on the Linux kernel for underlying functionality, such as threading and low-level memory management.

Further, we discuss the major design choices in the Dalvik virtual machines, such as the `.dex` format, Zygote, the register-based architecture and security.

**Dex file format.** In standard Java environments, source code is compiled into Java bytecode, which is stored within **.class** files (i.e. each Java class results in one `.class` file) and read by the virtual machine at runtime. On the Android platform, these `.class` files are converted into Dalvik executable (`.dex`) files. Whereas a `.class` file contains a single class, a `.dex` file holds multiple classes. Such `.dex` files are optimized for memory usage and their design is primarily driven by sharing type-specific constant pools. A constant pool stores all constant values used within a class and includes string constants, as well as field, variable, class, interface and method names. Rather than storing these values throughout the class, they are always referred to by their index in the constant pool.

In traditional Java classes, each class has its own private, heterogeneous pool, whereas in Dalvik many classes share the same type-specific pool and constants duplication is eliminated. Consequently, a `.dex` file contains significantly more logical pointers or references than a `.class` file. These memory sharing optimizations make sense in the context of Java classes, where more than 60% of the `.class` file size represents the constant pool, while the remaining corresponds to the method part. Additionally, garbage collection strategies must respect the memory sharing. However, garbage collections are independent between applications, although they are sharing memory blocks. To overcome these limitations, Dalvik keeps mark bits that indicate that a particular object is reachable, and therefore should not be garbage collected.

**Zygote.** Since every application runs in its own virtual machine instance, the bootstrapping process of these instances must be fast and the memory footprint must be minimal. Android uses the Zygote concept to enable both code sharing across instances and to provide low startup times for new instances. The Zygote design assumes that a significant number of core library classes and corresponding read-only heap structures are used across many applications. This refers to classes and data that most applications use, but never modify. These characteristics are exploited to optimize memory sharing across processes. In comparison, in the traditional Java virtual machine design each instance owns an entire copy of the core library class files and any associated heap objects, which means memory is not shared across such instances. The Zygote represents a virtual machine process

that, when initialized at system boot time, starts a Dalvik instance and preloads core library classes. Generally, these classes are read-only and therefore represent a good candidate for preloading and sharing across processes. Once the Zygote is initialized, it waits for socket requests from the runtime process, indicating that new virtual instances should be forked. Cold starting virtual machines requires longer times and can be an obstacle for isolating each application in its own instance. By spawning new processes from Zygote, the startup time is reduced.

**Register-based architecture.** Traditionally, virtual machine developers have favored stack-based architectures over register- based architectures (SGBE05), because of the ease of writing a compiler back-end and the code density, that generates invariable smaller executables for stack architectures. However, the simplicity and code density features impact performance. Studies on ARM devices have shown that a register-based architecture requires on average 47% less executed instructions that the stack-based (Van10). On the other hand, the register code is 25% larger than the corresponding stack code. Standard benchmarks show that register-based virtual machines require on average 32% less execution time. Given that Android is running on devices with constrained processing power, the Dalvik virtual machine is register-based. Although register-based code is larger than the stack-based, the significant reduction in the code size achieved through shared constant pools in the `.dex` file offsets the increased code size, means that users can still benefit from memory usage gains when compared to the Java virtual machine and the `.class` format.

**Security.** The Android architecture ensures that no application has permission to perform any operations that would directly impact other applications, the operating system or the user. This includes reading or writing the user private data or another application's files, performing unauthorized network access or keeping the device awake. These features are implemented across the operating system and framework levels. Each application is run within its own instance of the virtual machine, which itself is running in its own process. Each process is assigned a user ID that can only access a limited set of features in the system and its own data. If additional security permissions are required, the application can request them through its `Manifest` file. These permissions are verified at application installation time by the framework and are enforced by the operating system at runtime.

To address memory constraints and achieve fast bootstrapping, Dalvik securely shares core, read-only libraries between virtual machine processes, by allowing processes to read code, without any editing. Since applications cannot interfere with each other based on the operating system enforced security and the virtual machines are confined to a single process, Dalvik is not concerned with runtime security. The Android distribution maintains most of the standard Java security classes, which are used within the application space, without being configured for interprocess security.

#### 2.3.1.2 Building Applications

To build applications, the software developer kit (SDK) provides access to the application programming interface and includes a phone emulator, which duplicates the features and functions of a mobile device running on the Android platform. A difference from other mobile platforms is that with Android developers can create complex applications running in the background of other applications.

All applications are divided into four basic building blocks, as follows

- *Activities* – An activity represents that part of an application displaying a screen to the user. For example, a map application could have a basic map screen, a trip planner screen and a route overlay screen, which count as three activities. Although these activities work together to provide a cohesive user experience, each one is independent of the others.

- *Intents* – Intents represent the mechanisms for switching between activities. With the map example, if a user wanted to plot a trip, an intent would interpret the input and activate the route overlay screen.

- *Services* – A service is a component that runs in the background to perform long-running operations or to perform work for remote processes, without providing a user interface. For example, a service might play music in the background while the user is interacting with a different application, or it might fetch data over the network without blocking user interaction with an activity.

- *Content providers* – A content provider manages a shared set of application data. For example, the Android system provides a content provider that manages the user's contact information. As such, any application with the proper permissions can query part of the content provider to read and write information about a particular person.

### 2.3.2 Maemo

Maemo (Mae05) is an alternative to the Android mobile platform and is based on the Debian Linux distribution for ARM architectures. It mostly comprises of open source code and has been developed by Nokia in collaboration with projects as the Linux kernel, Debian and GNOME. Maemo uses an X Window system-based graphical user interface, namely X11, while the GTK-based Hildon framework functions as the application framework, as shown in Figure 2.4.

**Base distribution.** Maemo builds on GNU/Linux for the operating system core, as well as the file system, and GNOME/GTK+ for user interface architecture. It uses the same component packaging system as the Debian `dpkg` tool binary packages. New packages can be installed, existing ones can be easily

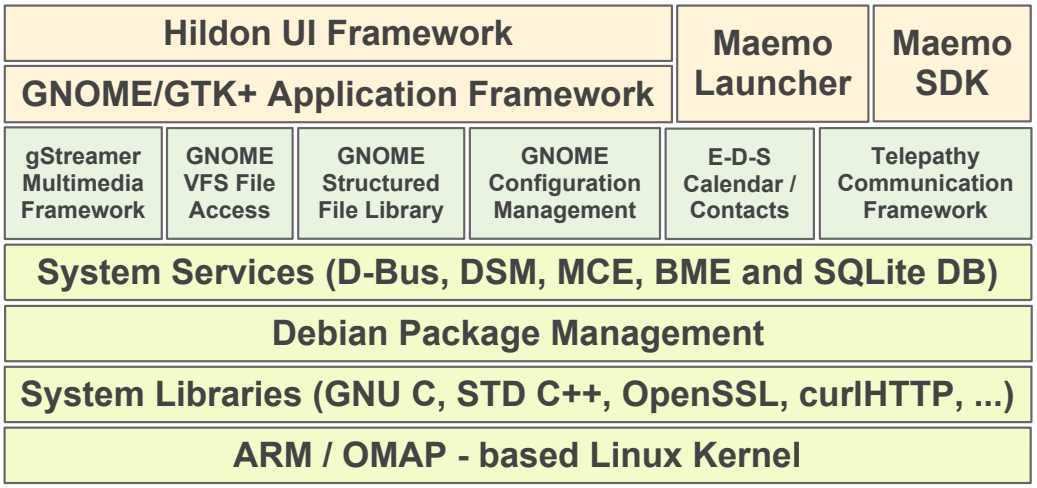



**Figure 2.4:** Maemo architectural overview.

removed and the whole system can be upgraded by the package management framework. In order to run the software on an Internet tablet, such as Nokia N810, various optimizations and enhancements have been made. These include power management related issues, touchscreen input, performance and size optimizations.

Instead of a multi-user system such as traditional Linux desktops, Maemo is considered a single user desktop system. The security model is focused on protecting the user from remote attacks, but not from other users. The most significant difference in the base system is given by the fact that Maemo uses a lightweight BusyBox replacement for the essential GNU utilities. In Maemo, the kernel and file system reside in separate partitions and cannot be updated with a package manager as with common Linux desktops.

**Application framework.** The purpose of the Hildon application framework is to provide support for application development by offering standard structures for building applications. For instance, applications with graphical user interfaces tend to have a similar event-driven runtime model. Hildon is partially based on the same technologies that GNOME framework is built on, most notably the GTK+ components. Also, it has several additions and enhancements to GNOME/GTK+ including the Hildon widget set, Sapwood theme engine and image server, task navigator, Hildon control panel and status bar. Some of these changes like Sapwood, for instance, are to reduce memory requirements and to improve speed on small devices. In addition, the Hildon framework has many features for supporting mobility like automatic state saving, touchscreen input methods and window management on such devices. The interprocess communication is managed by D-Bus messages. The user files are accessed through GNOME-VFS and multimedia applications can use GStreamer for accelerated

support for various codecs.

**System services.** The GNOME VFS file system enables applications to use a vast number of different file access protocols without requiring any underlying details. Some examples of the supported protocols are the local file system, HTTP, FTP and OBEX over bluetooth. In practice this means that all file access methods are transparently available for both developer and end user. The interface used for file handling is considerably more flexible than the standard platform offerings, since it features asynchronous reading and writing, MIME type support and file monitoring.

GConf is used by the GNOME desktop environment for storing shared configuration settings and preferences for applications. The daemon process `gconfd` follows the database changes, such that the new settings are applied to all applications using them. However, if specific settings are required by single applications only, the GLib utility for `.ini` style files should be used instead. For interprocess communications, Maemo relies heavily on D-Bus. The D-Bus framework allows programs to export their programming interfaces so that other processes can call them without having to define custom interprocess communication protocols. Using these exported programing interfaces is also language agnostic. A Maemo specific library called `libosso` provides helpful wrappers for D-Bus communication, application state saving and allows applications to be initialized, as well as to listen to system hardware state messages, such as "battery low".

**Development environment.** Supported Linux distributions for Maemo's development kit (SDK) are Debian and Ubuntu, but installing the environment is also possible for other distributions. The SDK creates a sandboxed development on a GNU/Linux desktop system largely built on the Scratchbox tool, such that cross-compiling is transparently handled. The development kit comes with architectures for two architectures. First, x86 is used for native performance and better tool support through native execution without need for emulation. Second, ARMEL is used for working with the actual device's architecture. Both have their benefits in the Maemo development and are provided as preconfigured targets inside a working Scratchbox installation. Generally, in active development the x86 environment is used because it provides the same performance as normal GNU/Linux applications. In addition, although the underlying architecture is different from the actual device, programs have the same behavior as if compiled and run on ARMEL. The default Scratchbox installation for cross-compiling works under most conditions and is quite accurate in emulating a full target environment on the mobile device.

**Programming model.** The programming model used from GTK+ is event-loop based, which means that the main loop sends events to one or multiple widgets whenever users perform actions. When widgets emit callback functions and their execution tasks finish, GTK+ returns to the main loop and waits for further events. The asynchronous model is useful when certain actions need to be

performed while the main loop waits for such events. Asynchronous actions are executed as non-blocking, meaning the control returns to the caller immediately and without interrupting the main program flow. Using asynchronicity makes the application interface more responsive and prevents the application "locking up" while, for example, reading data over the active network connection. For example the GNOME VFS programming interface has asynchronous counterparts to all functions. Callbacks for asynchronous operations are triggered in the normal event-loop, meaning that the application will be able to handle both user interface events and VFS events simultaneously. Another example of supporting asynchronicity is D-Bus.

Compared to Android, specifically designed for smart mobile devices and streamlined to their needs, Maemo is the closest experience to a full-blown operating system on hand-held devices. This means that most of the Debian packages could be compiled and run without adjustments, while the GDK as the windowing basis makes application development easier than with Android.

# Chapter 3

# AlfredO: Dynamic Software Deployment from Clouds to Mobile Devices

Applications are becoming more complex and their set of rich functionalities, as well as requirements, is rapidly expanding. These requirements cannot be satisfied on today's mobile devices alone. Even though the computing and communication capabilities of these platforms are improving, applications will continue to push platform limits as new, more compute-intensive functionalities are developed. This means that full installation and execution on the mobile device is not always possible. On the other hand, using devices as thin clients and leaving all the computation remote has the drawbacks of high data communication latencies and no customization. To overcome these limitations, in this chapter we propose a set of techniques that flexibly distributes an application in two parts, between the mobile device and a remote server or the cloud. The focus of our approach is on the client side and implicitly the user experience, therefore the goal is to minimize application interaction times and extend battery life on the device.

## 3.1 Motivation

Today's mobile users demand increasingly ubiquitous and complex applications on their devices (BZC06). 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 (BSPO03, FPS02, CBC+10, SBCD09, CM09), 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, 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 a user, enjoying a good network connection, starts uploading multiple photos of furniture items only to experience moments later a slower application responsiveness, due to varying network conditions (e.g., 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.

Another example comes from public transportation companies that use vending machines for ticketing to reduce costs. These machines become a bottleneck at peak hours and in busy stations. A significant part of the interaction – searching for the right ticket – could be offloaded from the machine to mobile devices. In this way, several customers can easily acquire tickets concurrently by browsing for the destination, type of ticket or connections on their devices. With the browser approach, users experience higher latencies due to the increased interaction frequency, which results in many connections, each adding the network latency delay. Installing such applications on the device prior to interaction is oftentimes not the best scenario, especially when there are many versions in all languages and for all locations. We argue that on demand code offloading with respect to variations in device capabilities or network connections, for instance, has the potential of providing users with better experience.

With our approach we address two important challenges, namely what parts of an application to offload and when, by considering the changing conditions one is likely to encounter when operating with mobile devices. We explore 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(OSG07, Apa10, RAR07) and various projects recognize the benefits of decoupling an application's functionalities into pluggable modules (Sil10, App11, Fra12, CM10). Thus, given a modular application, our approach delivers 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 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.

These concepts are implemented in AlfredO (GRJ+09, GRA12), a system that currently runs on Android (And07) and Amazon EC2 instances (EC2). An earlier version supported an environment consisting of the Maemo (Mae05) mobile platform and a remote server. We validate our approach through AlfredO with various applications and observe significant gains in both interaction time and battery consumption when comparing to the traditional approaches that offer no flexibility in application distribution.

## 3.2 Challenges

The goal of our approach is to dynamically distribute cloud-based applications between a mobile device and a remote server or virtual instance, such that the overall interaction time is minimized and battery life is improved. To match this goal we identify the following relevant challenges to address:

1. *Optimal application partitioning* – the decision on which part of an application should run remote and which on the mobile device is not obvious, but highly depends on the application structure and resource requirements, as well as specific platform constraints.
2. *On-the-fly application installation and updates* – in many real-life scenarios, it is not possible to assume that a mobile 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.
3. *Dynamic application partitioning* – mobile devices can experience changes in connectivity due to mobility and instability of wireless communication, as well as variations in the application load (both CPU and data transfers)

due to multiple concurrently-running applications. Our goal is to adjust the partitioning decision online and to reconfigure the current application deployment without interrupting ongoing interactions.

4. *Adaptation to varying data inputs* – the number and size of a user's data inputs (e.g. size of images to process, length of text to synthesize, etc.) can impact the application execution time and therefore its optimal partitioning between the mobile device and the remote instance. 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 remote side, our goal is to allow clients to autonomously decide which configuration to adopt once the user inputs have been submitted to the application.

## 3.3 AlfredO

Through our set of techniques implemented in AlfredO (RRA08, GRJ+09, GRA12), the thesis makes several important contributions. First, we show how to use an accepted software development technique to provide the basis for distributed deployment of services. This is important so that developers are not required to adopt new techniques in order to develop new services that can take advantage of a system like AlfredO. Second, we show how to modify a module management platform into a complete and automatic pipeline supporting the distributed deployment of applications between clouds and mobile devices. Third, we present an implementation that is capable of dynamically adjusting the distributed deployment of an application to optimize its behavior and performance according to several parameters. The dynamic aspects of AlfredO guarantee that on-the-fly acquisition of applications does not result in delays and that the system can adapt to the changing conditions one is likely to encounter when operating with mobile devices. Fourth, we introduce a novel mechanism to allow clients to autonomously and dynamically adjust an application configuration based on the user's inputs. Through these key ideas, our work takes an important step towards a better utilization of mobile device resources and an optimal deployment of applications, which in turn opens up many more possibilities than those supported by traditional approaches.

### 3.3.1 Architecture

To benefit from the AlfredO model, applications must be built in a modular fashion, where ideally modules contain highly-cohesive functionalities and communicate through low-coupled dependencies. The steps AlfredO takes to distribute a modular application between the cloud and the mobile device are shown in

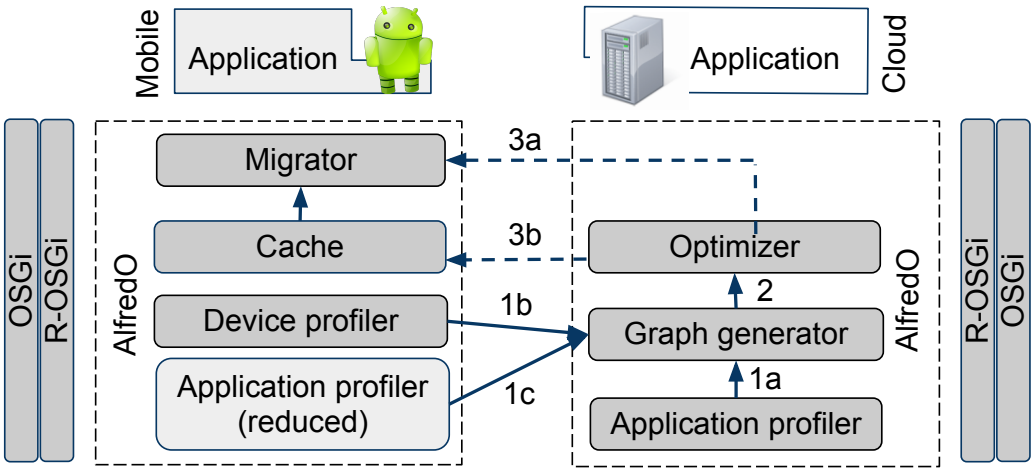



**Figure 3.1:** AlfredO architecture and pipeline of operators.

Figure 3.1. First, on the cloud side, the application profiler instruments the application to extract a compact description of its modular structure, as well as CPU and communication measurements (step 1a). On the client side, the mobile device profiler collects measurements of the CPU load, network status, power consumption and available storage space of the device (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 AlfredO optimizer identifies the optimal distribution of application modules and configures the deployment accordingly (step 3a). For different simulations of the user inputs, the optimizer computes the most suitable partitions and caches them on the mobile device, to allow the client to autonomously decide how to adapt the distribution when inputs vary (step 3b).

At bootstrap, AlfredO 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 devices CPU load by starting more applications) AlfredO's profilers and optimizer are constantly running such that the partitioning configuration can be changed on the fly.

The architecture of AlfredO is shown in Figure 3.1. The runtime on the cloud includes the application profiler, the consumption graph generator and the optimizer, which given that it is periodically executed, would become expensive on the client side. 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 client runs the device profiler and a reduced version of the application profiler collecting only CPU statistics.

Our system is implemented in Java and runs on the Android (And07) and Maemo (Mae05) mobile platforms. Both versions are based on the OSGi (OSG07) framework to manage the life-cycle management of application modules. However, different concrete implementations are used, namely Apache Felix (Apa10) for Android and Concierge (RA07) for Maemo, with R-OSGi (RAR07) on top to allow for remote execution across OSGi platforms.

We note here that due to the resource constraints of mobile devices and the specialized features of ubiquitous computing, the problem of software life cycle management has become more challenging and involves different trade-offs. First, resource limitations means that CPU cycles need to be saved, while memory should be carefully used. Second, Java virtual machines built for small devices are considerably more heterogeneous and less optimized than their desktop counterparts. While, these virtual machines are optimized to provide roughly equivalent performance on full-sized machines, for mobile versions this state of affairs does not hold. Many such virtual machines are optimized for size and consider performance as a secondary objective. For instance, optimizations tend to be hardware specific and dramatic changes in behavior can be observed with different devices.

Clearly generic solutions are preferred against hardware-specialized ones. In this context, Concierge (RA07) is designed without taking advantage of specific platform features and applies only generic optimizations. An important objective then becomes achieving a consistent behavior across platforms, given their heterogeneity. Additionally, Concierge provides Java code optimizations, avoids indirection and introduces the notion of average bundle. An average bundle captures application independent behavior and is used for base optimizations, while not considering user interaction effects or the effects of incoming calls from remote machines. Evaluation shows that Concierge has a considerably smaller file and memory footprint than existing implementations. It also improves performance by 2 – 3x factor compared to its counterparts.

While Concierge is not fully compatible with the latest mobile platforms, such as Android, new systems have emerged to alleviate this problem. On the one hand, Apache Felix (Apa10) manages the integration between OSGi and the Android mobile platform. While it is not specifically optimized for such resource-constrained devices, it provides developers with the capability of dynamically loading OSGi bundles by means of a few lines of code. On the other hand, highly optimized OSGi implementations have emerged lately and ProSyst (Pro11) is one of the few to tightly integrate the framework with the underlying operating system. It is compliant with the latest OSGi specifications and offers access from Android applications to OSGi services and vice-versa, by integrating Android intents and OSGi events. Additionally, ProSyst provides full mobile device management and allows one to control the OSGi runtime lifecycle through the user

interface. From this perspective, ProSyst could potentially replace Apache Felix in our system.

### 3.3.2 Guidelines for Practical Modularity

The principles of software modularity, discussed in Chapter 2.2, bring several important benefits. First, they allow for parallelization and independence during development, which in turn reduces the time and price of writing applications. Second, modularization supports object-oriented principles by encapsulating related information in a single container and exposing only the publicly supported interfaces to the rest of the application. The details of the module implementation remain hidden to external entities using the module. Third, by building an application as a collection of well-defined modules, one encourages the reuse of potentially large portions of code.

Plenty of research (BD11, SRK07, AG01) has been done to define general guidelines on how to modularize applications, as discussed in Chapter 2.2. Based on these, we formulate several practical rules that are relevant in our context, as follows:

*P1.* Application functional units *should* be layered– Every module within an application should have a well-defined goal and should provide a well-defined set of inputs and outputs, as well as clear interfaces to the specific services of higher-level modules. Adopting a layered architecture provides a good organizational concept of an application, in that all modules in a layer can only seek the services of the layers below.

*P2.* Application modules *should* have limited scope and similar code sizes – To maximize reuse, modules should have a limited scope. On the one hand, if a module expresses several disjoint scopes, its large size and excess functionality can discourage its reuse and complicates code evolution. On the other hand, a module should not be broken along artificial lines, simply to achieve smaller modules. Such an approach will most probably result in many dependencies, which create high costs for remote communication and reduce the flexibility of application distribution to fewer practical partitionings.

*P3.* Mutually dependent functionalities *must* reside in the same module (*cyclic dependencies between modules are not allowed*) – Mutual dependencies represent a particular category of module dependencies. First, we want to limit the number of dependencies per module, to enable code reuse and maintainability. Fewer incoming dependencies means that modules are easier to modify, without propagating changes to the entire application. Modules with fewer outgoing dependencies, on the other hand, are easier to reuse. Second, cyclic dependencies are not allowed. Let us consider two classes, A and B, whose methods call each

other at different moments in the execution timeline. If these classes would reside in different modules a cyclic dependency would be created. Such dependency is especially important when starting the application modules through OSGi and if present, no linear order can be determined and conflicts arise.

*P4.* Core functionalities *should* be separated from the user interface – Besides separating modules into tiers, this property additionally provides the flexibility of migrating the current user interface implementation to a native implementation. Further, changes within the core functionalities do not propagate in the user interface or vice-versa and such modules can be used independently by interacting through their APIs. The same separation principle is desired also between core functionalities and the data tier in the application. Separating the user interface from the remaining modules also benefits the startup time of the application on the mobile device, especially if no other modules are locally acquired.

*P5.* Determining how an application uses data – Understanding which part of the data is used either publicly (i.e. through the end user interface or API) or privately (i.e. within the internal core) can suggest how the applications should be modularized. An essential part of modularizing an application is to determine the end user requirements. This way, before development, one would already know which objects and methods need to be made visible through the user interface, and which can remain hidden from external services.

*P6.* Intermodule call traffic is API-based (non-API-based call traffic is not allowed) – This guideline results from encapsulation and ensures that module implementation details are hidden from other modules.

Oftentimes, modularizing large code bases means that certain application modules will implement more than one functionality, although logically related. An example is shown in Figure 3.3, where a module provides both encoding and decoding functions. In such scenarios, one should expose different services corresponding to each functionality, in order to allow higher-level modules to import only the needed service. The above properties provide an additional benefit to AlfredO, that of obtaining sparse graphs as ways of representing applications. This result is important to reduce search space in the optimization step. Let us consider that a large application has many, small functionalities that are strongly connected between themselves. As discussed later, the graph representation of such applications becomes dense and consequently the search space to find the optimal cut increases exponentially. This makes the optimization process in AlfredO significantly more complex and slower. In (GRJ+09) we show that with 50 bundles per application and with an average of four dependencies per bundle, a global optimizer needs over 100 seconds to find the best partition. Hence, when applications consists of many functional units, it is advisable to group functions with many inter-dependencies in the same module.

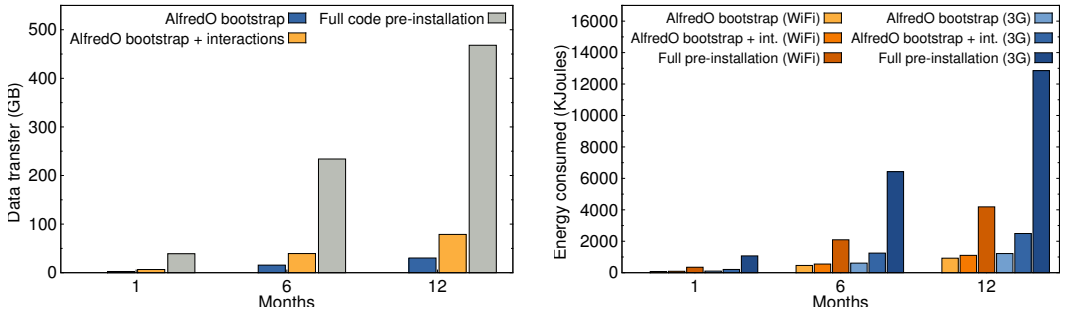
### 3.3.3 Code Pre-installation and Updates

In AlfredO we remove the need to pre-install an application on the mobile device before interaction and provide 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. We discuss this scenario through an example. Let us consider the FreeTTS text-to-speech synthesizer application (Fre10) (see Chapter 4.2.2) residing in the cloud and that clients download the application or update it as new versioned updates appear (i.e., every month). In a code pre-installation approach every update actually requires another full download since versioning-based updates are not supported. In AlfredO only the minimum code that enables user interaction will be offloaded. For our example, the compressed code base measures around 4MB, while with our approach only 260KB are sent. However, since the modularity principle supports versioning, when a new update is released only the newly updated modules will be sent, thus reducing the data transfer to less than 260KB. During user interactions with pre-installation no code needs to be migrated, so the data transfer will be 0. In our system, if external factors (i.e., CPU, user inputs, network bandwidth) determine changes in the best configuration, certain modules will need to be migrated. However, from our experiments with the text-to-speech synthesizer, as shown in Figure 3.2(a), at most an additional 415KB will be offloaded, because some modules are never convenient to run on the mobile device. Therefore, our approach is 15-fold more economical than pre-installation for bootstrap and 6-fold better when also accounting interactions. In fact, AlfredO can achieve even lower data transfers through versioned updates, which were disabled in this example.

Additionally, modularity allows us to naturally foster the deployment of applications that contain critical or security restricted pieces of code, and thus cannot by fully hosted on the mobile device (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.

Finally, we show that AlfredO consumes less energy than pre-installation approaches when considering the overall data transfer. Although there is no linear relationship between data transfer and energy consumption, we use a general energy model defined in (BBV09). For 3G, the transfer energy is 0.025(x) + 3.5, while for WiFi is 0.007(x) + 5.9, where x is the amount of transferred data in bytes. As shown in Figure 3.2(b), AlfredO is at least 4 times more energy efficient for WiFi transfers. For 3G, when accounting user interactions our approach consumes 5 times less energy, while for bootstrap only it is an order of magnitude




(a) Data transfer comparison

(b) Energy consumption comparison

**Figure 3.2:** Data transfer and energy consumption comparison between *AlfredO bootstrap*, *AlfredO bootstrap + interactions* and *Full code pre-installation* for 10000 clients, during 1, 6 and 12 months (WiFi and 3G are used).

more efficient. The measurements do not include the tale energy, because it is independent of the data amount transferred and can be resumed to a constant.

### 3.3.4 Flexible Bundle Deployment

Bundles are reusable pieces of software packaged in binary components, containing bytecode and metadata (i.e., versioning and explicit dependencies). Modular designs encourage the coupling of related functions in the same bundle, which, if available for sharing with other bundles, are exposed through a *service interface*. To ensure communication between different bundles, modularization introduces the notion of *dependencies*. Such dependencies from one application bundle to another assumes that the dependent bundle explicitly imports and references 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, similar to (RSDI05).

Since the OSGi model is restricted to single machines, AlfredO 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.

An application running with AlfredO provides a descriptor that contains information about all migrated bundles, such as exposed services, profile data or list of dependencies. Details about the application specification are provided in Chapter 3.3.6. The migrator component uses directives provided by Apache Felix

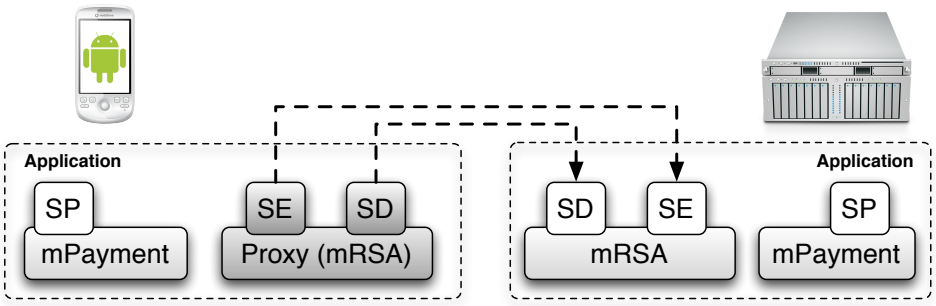



**Figure 3.3:** Example of bundle deployment using R-OSGi.

and R-OSGi, to provide support for the following operations:

1. parse the application descriptor to obtain the full list of services to keep in the local registry, such that an updated list of all currently available services on the mobile device is available;
2. parse the application descriptor to compute the order in which bundles need to be installed and started, as well as how services needs to be initialized, such that no conflicts arise (i.e., considering two bundles $M_i$ and $M_j$ such that $M_i$ calls methods from $M_j$'s service, a conflict would appear if $M_j$ is started after $M_i$);
3. service look-up in the local registry and service binding, such that actions or events initiated from other services or within the user interface code are invoked on the correct service through reflection;
4. periodically poll a method provided by a remote service, such that it can react to its changes by invoking another method;
5. dynamically migrate the code corresponding to an application bundle, as well as install and start it on the mobile device by means of standard Apache Felix directives;
6. generate the remote proxies for all remotely located dependencies of an application bundle with R-OSGi operations;
7. remove the remote proxies of a bundle once its code is acquired on the client side and rebuild the dependencies of all caller bundles.

When an end user wants to access one of the bundles provided by the application, the bundle service is shipped to the mobile device and a local proxy is generated, as well as registered with the local registry as an implementation of the particular bundle. If the types referenced in the service are standard in Java, then the operation is complete. Otherwise, if it references specialized types that

are provided by the original implementation, the corresponding classes need to be transmitted and injected into the proxy bundle. However, a possible enhancement to the simple delegation of service calls to the bundle located remotely is providing additional functionality to proxies. Such proxies can support the migration of parts of bundle functionality on the client side. This is achieved by providing an abstract class, such that implemented methods run locally on the device and any abstract methods are implemented as remote calls.

The migrator supports all the interaction patterns of both OSGi and R-OSGi systems, such as the life cycle management of services and bundles, the asynchronous communication through events, data transfers through stream proxies, as well as synchronous invocations between services. Given the service-oriented approach provided in AlfredO, where concrete implementations and remote proxies are hidden, local and remote services are interchangeable, such that a user cannot differentiate between them. Moreover, the deployment of an application that runs atop of AlfredO can be dynamically modified at runtime, by acquiring remote-located bundles on the mobile device, for instance.

In Figure 3.3, 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, AlfredO's goal is to switch between partitions without interrupting an ongoing interaction. To achieve this we exploit R-OSGi's feature of dynamic bundle management. Changing a configuration consists of 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 in our example that the optimizer decides to fetch also *mRSA* on the mobile device. We also assume that *mRSA* has dependencies on other bundles (not shown in the figure). To initialize *mRSA* on the client, AlfredO 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 the remote proxy for *mRSA* 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 user interaction, without the need for the current configuration to stop. However, operation d) usually takes a very short time (less than 1 second for the the applications we have so far considered), such that the application's downtime is negligible.

### 3.3.5 Instrumentation and Profiling

The first step in optimizing application deployments and adapting them when environmental changes occur, is to characterize the behavior and resource demands

of such an application. We assume applications to be built using the OSGi module system, but similar methods can successfully be applied with applications modularized using different tools.

#### 3.3.5.1 Offline Profiling

Initially, we instrument all modules composing the application in a static approach, to understand its behavior and execution path, as well as to measure its resource demands. In our typical scenario, the relevant resources represent the consumed CPU and memory on the mobile device, the code size to be fetched on the client and the data traffic generated at bundle level both in input and output.

The offline feature of profiling means the instrumented application needs to be executed a-priori on one or multiple mobile platforms, in order to collect on a debug channel the accurate amounts of consumed resources. However, once the application execution is initiated, no new measurements are gathered and all optimization decisions are based on the already existing statistics. The obvious advantage to such an approach is the lack of overhead on the underlying platform. However, offline profiling suffers from decreased accuracy especially when AlfredO needs to identify the best partitioning distribution if external factors (i.e., CPU consumption on the mobile device, network bandwidth, user inputs) vary.

#### 3.3.5.2 Online Profiling

To overcome the drawbacks of offline profiling, AlfredO performs online profiling, managed at the application level through code injection and collecting measurements at every user interaction. First, the profiler 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 (HXT$^+$10) and mobility makes network conditions even more unstable. For these reasons, the AlfredO profiler monitors on the mobile device which network interface is currently in use and what speed is achieved on the link.

**Application Profiling with Structural Reflection.** Application profiling is implemented using load-time structural reflection (Che00) at bytecode level. Every bundle has a MANIFEST file containing metadata about versioning, services

and dependencies on other bundles. For each service, the profiler identifies the actual Java classes implementing it and in all methods it injects code to measure the execution time and the size of I/O parameters. The overall execution time per bundle is the sum of the running times of all methods that are actually executed. Measuring the data transfer between bundles allows us to identify which bundles are closely coupled and can benefit from running on the same machine. The bundle execution time instead allows us to identify computational-intensive bundles which might cause performance degradation if executed 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 machine. At runtime, the profiler monitors only the dynamic variables. To avoid flapping in the profiler measurements, it maintains a history of measurements and computes exponentially smoothed moving averages.

**CPU Profiling.** Unlike systems such as CloneCloud (CIM+11), AlfredO does not assume offline profiling of the CPU time for every possible application and on every possible mobile platform. In fact, this would require running all application configurations for all considered applications on the mobile client, 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 very 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 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 the future, this device-specific parameter could be provided, for instance, by the application provider, together with the mobile platforms supported. 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 executing on the client. In addition, on the mobile side we periodically obtain the current CPU load of the device, over all active processes, from Android API functions.

**Network Profiling.** Network capabilities on the mobile device vary, sometimes dramatically, over short spans of time, as shown in Table 3.1, where we report download and upload speeds, as well as latencies, experienced on HTC Desire at different times during 12 hours. In the WiFi scenario, the download speed is always lower than the upload speed and it is confined between stable minimum and maximum values. For example, we notice close values for both download and upload during the 12pm – 6pm interval. However, in the 3G scenario, mea-

**Table 3.1:** WiFi and 3G download speeds, upload speeds and latencies experienced at different times during 12 hours (average and [standard deviation]).

| | | 10am | 12pm | 2pm | 6pm | 10pm |
|---|---|---|---|---|---|---|
| Download (kbps) | WiFi | 3464 [342] | 5217 [136] | 4811 [151] | 5366 [286] | 4522 [197] |
| | 3G | 337 [58] | 2398 [688] | 2977 [514] | 668 [172] | 2468 [537] |
| Upload (kbps) | WiFi | 5944 [262] | 8194 [247] | 8105 [133] | 8741 [249] | 6471 [228] |
| | 3G | 1111 [342] | 921 [332] | 865 [377] | 1174 [288] | 1193 [371] |
| RTT (ms) | WiFi | 30 [10] | 33 [3.6] | 28 [5] | 25 [8] | 27 [6] |
| | 3G | 130 [16] | 137 [14] | 143 [21] | 139 [15] | 135 [16] |

surements performed at 10am and 6pm show higher values for the upload speed, while the opposite is true for all other cases. It is clear that while the standard deviation for 3G is mostly lower than for WiFi, the 3G connection is less stable because neither the download nor the upload speeds have monotone values. The obtained results are supported by the experimental comparison in (GD10), which shows that WiFi performs comparably well against 3G when downloading and even significantly better while uploading data. In addition, we observe a lower latency for WiFi connections when compared to 3G, which confirm the claims made in (CBC$^+$10) stating that latency is an important impacting factor in 3G networks.

In order to test the downlink and uplink performance, there are several tools available for Android, such as SpeedTest (Spe06) and MobiPerf (Mob11). Although they provide extensive additional information, like ping latency, signal strength, TCP handshake latency or DNS lookup latency, both lack flexibility in allowing the user to select servers located in specific geographical locations. For example, for users located in Switzerland, SpeedTest provides a list of available servers within a 700km radius, while MobiPerf uses a single server located outside of US. Given that in our evaluation we considered Amazon EC2 virtual instances located in the US, in addition to nearby remote servers, the performance tests must be able to use servers located in the same regions. Therefore, since neither SpeedTest or MobiPerf were suitable to measure the ping latency and downlink, as well as uplink speeds, we implemented our own client – server application and deployed the server code on rented EC2 US instances. We use a similar methodology as described in (Mob11), such that the client communicates to the server running on the virtual instance and transfers data packets for 15 seconds. Packet traces are collected at the server side. The downlink and uplink server are running on ports 5001 and 5002, which are verified as not blocked by most carriers. The duration of 15 seconds is chosen such that TCP can pass its slow-start stage and reach a saturation point. To measure the connection latency, we simply execute

**Table 3.2:** WiFi and 3G download and upload throughput experienced while transferring 10KB, 50KB, 100KB, 200KB, 500KB, 1MB, 2MB and 3MB worth of data.

| Payload size | WiFi | | | | 3G | | | |
|---|---|---|---|---|---|---|---|---|
| | EC2-Small | | EC2-Large | | EC2-Small | | EC2-Large | |
| | Down | Up | Down | Up | Down | Up | Down | Up |
| 10KB | 0.337 | 0.317 | 0.219 | 0.198 | 0.8 | 0.43 | 0.18 | 0.41 |
| 50KB | 0.735 | 0.551 | 0.43 | 0.458 | 2.12 | 1.16 | 0.38 | 0.8 |
| 100KB | 1.024 | 0.718 | 0.67 | 0.65 | 7.98 | 4.07 | 1.34 | 2.79 |
| 200KB | 1.734 | 1.546 | 0.919 | 0.975 | 12.24 | 6.41 | 2.41 | 4.66 |
| 500KB | 4.903 | 4.313 | 1.76 | 1.763 | 40.29 | 23.97 | 3.11 | 6.07 |
| 1MB | 7.264 | 5.238 | 3.248 | 2.659 | 67.88 | 35.43 | 8.19 | 15.94 |
| 2MB | 10.019 | 8.483 | 6.024 | 5.067 | 90.18 | 48.34 | 10.14 | 18.9 |
| 3MB | 14.029 | 12.364 | 8.836 | 7.025 | 138.27 | 74.98 | 16.53 | 30.47 |

a `ping` command to the public IP address of the virtual instance from the HTC Desire device.

Motivated by download and upload speeds, as well as the latencies experienced with WiFi and 3G, we attempt to estimate the data transfer times between communicating modules by understanding how these factors are reflected in ground measurements. Therefore, considering an HTC Desire device and EC2 small or large US instances, we measure the time required to transfer back and forth 10KB, 50KB, 100KB, 200KB, 500KB, 1MB, 2MB and 3MB worth of data with WiFi and 3G (Table 3.2). In the WiFi scenario, when using both EC2 small and large US instances, the RTT latency is in the range of 25 – 37 ms. The download speed varies around 2000 – 2200Kbps and around 4000Kbps in the EC2 small and large experiments, respectively, while the upload speed varies around 2300 – 2500Kbps and 4500Kbps. However, for 3G the variations experienced are more significant, with the upload speed varying around 609Kbps during the EC2 small experiment and around 1100 – 1200Kbps for large instances. We observe an even larger variation for the download speed, from 300 to 2000Kbps. The RTT remains relatively stable, with values in the range 95 – 110 ms. In order to measure accurate values for the download and upload throughputs, both the mobile device and the EC2 instance need to be time synchronized. To ensure this, we run the `ntpd` daemon on the virtual instance and an NTP service on the HTC Desire, with both synchronizing every 15s to the `pool.ntp.org` server.

Based on these experiments, we define a simple model that is able to estimate

with relatively high accuracy the download and upload speeds for different data sizes, as follows:

$$x_{speed} = \frac{payload}{Time_x - \frac{RTT}{2} - \frac{RTT * payload}{2 * TCP_{window}}} \tag{3.1}$$

where $Time_x$ represents the estimated median download or upload time for transferring data with size given by *payload*. We recall here that the amount of data received in input and produced in output by an application modules is extracted in the application profiling step. Given that all experiments are performed through TCP connections, our model accounts for the initial connection latency as $\frac{RTT}{2}$ and the window size $TCP_{window}$, which determines how many packets are sequentially sent to transfer the entire payload. On the HTC desire device, the TCP default window size is 64KB for download and upload transfers, while both EC2 small and large instances provide the same window size of 128KB. This means the bottleneck is the device and the model accounts for the 64KB window size.

We apply our model in the network profiler in two phases. First, before any user interaction with the application has been initiated, the network is pruned by sending periodically (i.e., every 30 seconds) data packets of 50KB from the mobile device to the remote instance that is connected with, in addition to pinging in order to obtain the current latencies. Once an application interaction starts, *opportunistic profiling* is used instead, such that the network speed estimation is based on transfers carrying actual application data. Additionally, on the mobile device, the AlfredO profiler detects whether the user is using WiFi or 3G by parsing the content of `proc/net/dev`, which is made available in the Android operating system.

Alternative approaches to ours could be used to profile the network bandwidth. Several such tools have been described in (YL03) and classified based on their measurement techniques, as well as on their focus on one or more of three related metrics: capacity, available bandwidth and bulk transfer capacity. Such examples are Pathload (Pat06), IGI (HS03) or pathChirp (RRB+03). In (KN01), the authors discuss several approaches to measuring the network latency, based on static filtering (NSN+97) or active probing (BRS99, LB00, Kes91). Any of these approaches could be used in AlfredO to estimate network latency.

**Power Consumption Profiling.** In order to profile the power consumed by running application bundles on the mobile device, AlfredO uses PowerTutor (Pow09), an online power and battery behavior estimation system that has been implemented for the Android platform. PowerTutor uses an automated model described in (ZTQ+10) to estimate how much of the device power is consumed by CPU, LCD display, GPS, WiFi and cellular networks. The accuracy of the model has been evaluated with popular applications, such as YouTube, Google Maps

or Google Talk, and shows that the average long-term error is less than 2.5% over the application lifespan and the average error is less than 10% for 1 second intervals. Additionally, the authors have measured the power overhead of their online estimation technique and found that it is an order of magnitude lower than the power consumption of high-power states of most smartphone components.

Since AlfredO considers CPU and network as prime factors for application bundle distribution, we profile the CPU and WiFi/3G statistics provided by PowerTutor and define the power consumption as follows:

$$Power_{consumed_{device}} = Power_{WiFi|3G} + Power_{CPU_{device}} \tag{3.2}$$

The purpose of measuring the power consumed for an ongoing application is to validate whether our latency-based model is not only effective in minimizing the overall interaction time, but also in lowering the power consumed on the mobile device. Therefore, the requirement is for the power consumption in the optimal configuration (`Opt`) found by AlfredO to be smaller than those experienced with traditional approaches:

$$Power_{\texttt{Opt}} < Power_{\texttt{UI}} \tag{3.3}$$

$$Power_{\texttt{Opt}} < Power_{\texttt{All}} \tag{3.4}$$

where `All` means running the application entirely on the mobile device, and *UI* means acquiring only the user interface locally and keeping all the computation in the cloud.

### 3.3.6 Application and Network Specification

Using the profiled data, AlfredO provides a compact description of each application module. A `module` is a logical unit encompassing one or more application functions. In the Java context, for instance, a module is packaged as a JAR file, containing a set of resources (i.e., compiled classes, libraries) and a manifest file describing the module. Below we show an example of module specification (lines 1–7) and statistics about network connectivity provided by the profiler.

```
1. module 'mPayment' {
2.    dependencies: [mRSA, mBank]
3.    type: [nIO, movable]
4.    CPU: 168ms
5.    size: 17kB }
6. wire 'wPR': ('mPayment', 'mRSA', 50KB)
7. wire 'wPB': ('mPayment', 'mBank', 7KB)

8. network: ('WiFi', 6Mbps, 2Mbps)
```

A module, such as the mPayment one used in the ticket machine application described in Chapter 4.2.2, has a number of `dependencies` 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 connections or 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` holds information about the type of network connection the device is currently using (WiFi or 3G), as well as its measured download and upload speeds.

### 3.3.7 Consumption Graph

The output of the profiling and specification steps is used to represent the application as a directed acyclic graph $G = \{\mathbb{M},\mathbb{E}\}$, where every vertex in $\mathbb{M}$ is an application module $M_i$ and every edge $e_{ij}$ in $\mathbb{E}$ represents a service dependency between $M_i$ and $M_j$. We use the notions of module and bundle interchangeably from here on. Each bundle $M_i$ is characterized by the following parameters:

- *type*: movable or non-movable bundle,
- $resource_i$: resource that characterizes $M_i$, such as the memory consumption on a mobile device platform ($mem_i$) or its execution time ($time_i$),
- $c_i$: the size of the compiled code of $M_i$,
- $in_{ji}$: the amount of data that $M_i$ receives as input from $M_j$,
- $out_{ij}$: the amount of data that $M_i$ transfers as output to $M_j$.

Figure 3.4 shows an example of a graph consisting of six bundles. We call this an application's *consumption graph*. Notice that although our implementation currently considers the above parameters to characterize the application, the model is generic enough to be easily extended with more variables.

We note that every application bundle can expose one or multiple services, which means the mapping between a bundle and its services is 1:m. In real scenarios, if an application bundle implements various functionalities that can be easily separated (e.g., example of *mRSA* bundle in Chapter 3.3.3), it is recommended to provide several services. The benefit of this practice is that it respects the principles of code reuse and coupling, in that every depending bundle will need to import only the service that implements a specific type of functionality.

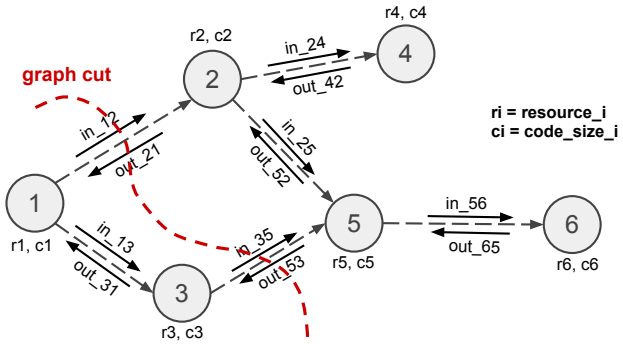



**Figure 3.4:** Example of application consumption graph.

Additionally, we have observed that the size of an application after modularization is in order of a few tens of bundles. Although there exist frameworks like Eclipse (Ecl12) that have significantly more modules, for ubiquitous, mobile-driven applications the set of functionalities provided can be decomposed in fewer bundles according to the principles of modularity.

### 3.3.8 Optimization Algorithms

The partitioning problem seeks to find a cut in the consumption graph such that some application bundles execute on the mobile device side and the remaining ones remotely, either on a server or on cloud virtual instances. The optimal cut maximizes or minimizes an *objective function* $O$ and satisfies a set of device resource constraints in the same time. The objective function expresses the general goal of an application partitioning. This may be, for instance, to minimize the end-to-end interaction time between a device and a remote instance, minimize the amount of exchanged data, or complete the execution in less than a predefined time. In AlfredO, the partitioning goal is to reduce the overall interaction time of the application.

A set of device-specific constraints may include $memory_{MAX}$, the maximum memory available for all potentially acquired bundles, or $code\_size_{MAX}$, the maximum amount of bytes of compiled code a device can afford to transfer from the server side. By default, our constraints restrict the maximum storage space available for installing the acquired modules and the maximum amount of data which can be uploaded to the cloud or remote server, or the maximum amount of memory to be consumed by local bundles.

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. AlfredO's default 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.

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

$$if:\ \sum_{i=1}^{k} c_i \leq C_{MAX}\ \ and\ \ \sum_{i=1}^{t \leq k} \sum_{j=1}^{w \leq l} in_{ij} \leq D_{MAX}\ \ or\ \ \sum_{i=1}^{k} mem_i \leq mem_{MAX} \quad (3.5)$$

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 ($t_{is} * k$) and generating the proxies for all remote dependencies ($t_p * r$). In order to become active, modules need to be installed and started, and proxies must also be established to manage the communication client-cloud. 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 mobile 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. In our experiments, we found the time for starting one 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, as well as the time necessary for transferring data between the distributed modules. Depending on the type of interaction, an application module may be invoked multiple times. To capture this information, we consider the parameter $f_i$ which specifies how many times the application module $M_i$, either on the mobile device or remotely. Additionally, $f_{ij}$ represents how many times the communication between $M_i$ and $M_j$ occurs.

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 (KK98) 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 (BDR+07) 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.

Before running the actual algorithms, we preprocess 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 the consumption graph $G = \{M, E\}$, if the cost of an edge $e_{ij} \in E$ is such that $in_{ij} + out_{ji} > D_{MAX}$, then $M_i$ and $M_j$ are merged into one bundle $M_{ij}$: all input and output edges are updated accordingly, and the cost of the new bundle $M_i$ is given by the sum of the relative costs of the initial $M_i$ and $M_j$.

#### 3.3.8.1 ALL

After the preprocessing 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 precomputed offline or as a dynamic problem where the partition must be calculated on-the-fly, once a mobile connects and communicates its resources. We consider both options and we propose `ALL` for offline optimization and `K`-step for online optimization.

The `ALL` algorithm, shown in Algorithm 1, always guarantees to find the optimal cut. It operates in three steps. First, it generates all "valid" configurations. Given $M$, 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 defined as follows: if the bundles $M_i$ and $M_j \in P_{device}$ are not connected by a direct dependency edge $e_{ij}$, then all bundles on the possible paths between $M_i$ and $M_j$ also belong to $P_{device}$.

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

1. $\sum_{i=1}^{k} mem_i \leq mem_{MAX}$ or $\sum_{i=1}^{t \leq k} \sum_{j=1}^{w \leq l} in_{ij} \leq D_{MAX}$

**Algorithm 1** ALL algorithm.

**Input:** $\mathtt{O}, \mathtt{P}, mem_{MAX}, C_{MAX}, D_{MAX}$
**Output:** $\mathtt{P}_{device}$, $\mathtt{P}_{cloud}$

1: *Initialize* C, FC *to* $\emptyset$
2: $C \leftarrow$ `getValidConfigurations(M, E)`
3: **for** $\mathtt{C}_c \in$ C **do**
4: $mem_{C_c} \leftarrow$ `getMemOnDevice(`$\mathtt{C}_c$`)`
5: $code_{C_c} \leftarrow$ `getCodeOnDevice(`$\mathtt{C}_c$`)`
6: $d_{C_c} \leftarrow$ `getDataTransfer(`$\mathtt{C}_c$`)`
7: **if** $((mem_{C_c} \leq mem_{MAX}$ `or` $d_{C_c} \leq D_{MAX})$ `and` $code_{C_c} \leq C_{MAX})$ **then**
8: *Add* $\mathtt{C}_c$ *to* FC
9: **end if**
10: **end for**
11: $\mathtt{P}_{device} \leftarrow$ `getMin(FC, O)`
12: $\mathtt{P}_{cloud} \leftarrow \mathtt{P} - \mathtt{P}_{device}$

2. $\sum_{i=1}^{k} c_i \leq C_{MAX}$

Third, the algorithm evaluates the objective function for each valid configuration and chooses the one providing its minimum value. The algorithm complexity is $O(|M| * |E| * log|E|)$, where $|M|$ and $|E|$ represent the cardinalities of the modules and dependency links sets.

#### 3.3.8.2 K-Step

While the ALL algorithm inspects all possible configurations and identifies the global optimal partitioning, the K-step algorithm, shown in Algorithm 2, evaluates a reduced set of configurations and finds only a local optimum. The K-step algorithm is therefore by design faster than the ALL algorithm, but also prone to be less accurate.

Instead of generating all configurations and then choosing the best one, 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 smaller than the current one and if the mobile device'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.

**Algorithm 2** K-Step algorithm.

**Input:** $\mathtt{O}, \mathtt{P}, \mathtt{K}, memory_{MAX}, C_{MAX}, D_{MAX}$
**Output:** $\mathtt{P}_{device}$, $\mathtt{P}_{cloud}$

1: *Initialize* $\mathtt{P}_{device}$, $\mathtt{Q}$, $\mathtt{C}$ *to* $\emptyset$
2: $\mathtt{root} \leftarrow \mathtt{getRoot(M, E)}$
3: *Add* root *to* $\mathtt{P}_{device}$
4: $\mathtt{O}_{init} \leftarrow \mathtt{compute}(\mathtt{P}_{device}, \mathtt{O})$
5: $\mathbb{K} \leftarrow \mathtt{getModulesAtKHops(K,}\ \mathtt{P}_{device}, \mathtt{M, E})$
6: $C \leftarrow \mathtt{getValidConfigurations}(\mathtt{P}_{device}, \mathbb{K})$
7: **while** $Q <> \emptyset$ **do**
8:  **for** $\mathtt{C}_c \in \mathtt{C}$ **do**
9:  $memory_{C_c} \leftarrow \mathtt{getMemOnDevice}(\mathtt{C}_c)$
10:  $code_{C_c} \leftarrow \mathtt{getCodeOnDevice}(\mathtt{C}_c)$
11:  $d_{C_c} \leftarrow \mathtt{getDataTransfer}(\mathtt{C}_c)$
12:  **if** $((memory_{C_c} \leq memory_{MAX}$ `or` $code_{C_c} \leq C_{MAX})$ `and` $d_{C_c} \leq D_{MAX})$ **then**
13:  $\mathtt{O}_{C_c} \leftarrow \mathtt{compute}(\mathtt{C}_c, \mathtt{O})$
14:  **if** $(\mathtt{O}_{C_c} < \mathtt{O}_{init})$ **then**
15:  $\mathtt{P}_{device} \leftarrow \mathtt{C}_c$
16:  $\mathtt{O}_{init} \leftarrow \mathtt{O}_{C_c}$
17:  $\mathtt{update(Q, C)}$
18:  $\mathbb{K} \leftarrow \mathtt{getModulesAtKHops(K,}\ \mathtt{P}_{device}, \mathtt{M, E})$
19:  $\mathtt{C} \leftarrow \mathtt{C} \cup \mathtt{getValidConfigurations}(\mathtt{P}_{device}, \mathbb{K})$
20:  **end if**
21:  **end if**
22:  **end for**
23: **end while**
24: $\mathtt{P}_{cloud} \leftarrow \mathtt{P} - \mathtt{P}_{device}$

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 device 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 device constraints is found. If none of the new configurations provide a better objective value, while satisfying the device 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. The complexity of the algorithm is $O(K * |M| * log|E|)$.

### 3.3.9 Dynamic Adaptation of Partitions

Since the execution environment can change dynamically (e.g., changes in CPU load on the mobile device or the network bandwidth) AlfredO needs to be able to promptly switch from an application distribution to another, if necessary. In order to achieve this, the optimizer periodically runs and detects when the current partitioning configuration 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 cost of the bootstrap phase, AlfredO 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, AlfredO installs the newly fetched modules. Once the initialization of the new configuration has finished, if there is an ongoing interaction, at its termination AlfredO seamlessly switches to the new configuration which will be used from the next interaction onwards.

#### 3.3.9.1 Adaptation to CPU and Network Variations

When variations in the CPU load or network bandwidth on the mobile device occur, in most cases the active configuration used for the application execution becomes sub-optimal, and therefore needs to be replaced with another scheme that provides a lower interaction time.

In the scenario of increasing CPU consumption on the mobile platform, one would expect that the execution time of the application bundles running locally to become larger. Given that our objective function accounts for bundle-level execution times, the AlfredO optimizer would decide to switch to a distribution configuration that enables most of the application to run remote. However, in the scenario of decreasing CPU load, it is not correct to observe that the opposite would happen (i.e., the optimizer would switch to a configuration that fetches

more bundles on the mobile device). In fact, the optimizer's decision is more complex and correlates the CPU drop with the variation in network bandwidth, especially the upload speed of the mobile device. Therefore, the optimizer decides to fetch more application bundles on the mobile device only if the data traffic, and consequently the communication cost, decreases compared to keeping the active distribution.

#### 3.3.9.2 Adaptation to Varying User inputs

Besides variations in CPU load and network bandwidth, a user's data inputs can significantly affect the partitioning decision. This is relevant for a large class of interactive applications that AlfredO 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 in input 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 solution with the current network conditions and some default user inputs. It then generates additional solutions by simulating other possible operating scenarios. Scenarios are defined by varying various *features*, describing both the operating environment and user inputs. For instance, the *network* feature has the format *network(lower_bandwidth, upper_bandwidth)* 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 pull 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.

## 3.4 Summary and Discussion

With the the functionality of mobile applications ever increasing, designers are often confronted with either the computational or network limitations of devices. As pointed out by recent trends, application partitioning between mobile devices and clouds have the potential of providing solutions for some of these issues, while improving performance and/or battery life. We argue that the static decisional techniques proposed in existing work cannot leverage the full potential of application partitioning, nor can it provide users with optimal interactions when environmental factors change. Thus, to allow for variations in the execution environment, we have developed AlfredO, a system that dynamically adapts the application partitioning decisions. The system works by continuously profiling an application's performance and dynamically updating its distributed deployment to accommodate changes in the network, data loads and the CPU utilization on the device. Unlike other systems, AlfredO removes the assumptions that code and data is always present on the mobile device, as well as on the server or cloud side. We argue that considering these dynamic code and data transfers represents a more realistic scenario in the mobility context, but also makes the optimization problem more challenging.

To optimally distribute an application between the mobile device and a cloud virtual instance or a remote server, we consider the objective of minimizing the overall application interaction, while accounting for all code and data migrations that might be required. To solve the problem, two algorithms are proposed, namely `ALL` and `K-Step`. The `ALL` algorithm always finds the global optimum partitioning. However, due to its higher computational cost implied, it is better suited for offline decisions. On the other hand, the `K-Step` algorithm improves the quality of its local optimum distribution as `K` increases, making it better suited for online decisions and especially with lower values of `K`. Additionally, to adapt to user inputs, we propose a caching technique that dynamically estimates the best partitioning based on input ranges and provides decisional autonomy to the mobile device.

Although AlfredO assumes that applications are modularized and the principle of modularity – loose coupling and high functionality cohesion – has been

gaining momentum recently, there are still no definitive guidelines on how to best build such modular applications or modularize existing ones. In the next chapter, experimental results show that even with applications that have not been modularized by an expert, AlfredO can provide significant gains in both performance and power consumption on the device.

In order to optimally distribute an application between the mobile device and a remote server or cloud virtual instance, AlfredO requires a-priori accurate measurements of the application execution and data transfers for each specific setup. In practice, this means running every application in any setup and with all possible distribution partitionings. However, given the multitude of mobile devices and, for instance, cloud virtual instances, making such an assumption is unreasonable. Therefore, to address AlfredO's limitation, we require a model that is able to accurately estimate the performance of an application for specific execution environments, without gathering extensive statistics before user interaction. We propose and evaluate such a performance model in Chapter 5.

# Chapter 4

# AlfredO: Applications and Experimental Results

This chapter evaluates how our approach integrates cloud-based applications on mobile devices through distribution, such that their overall performance and the power consumption on the client is reduced. Additionally, we discuss how the proposed techniques, implemented through AlfredO, are able to provide a better utilization of resources on mobile devices and to offer flexibility in deploying such applications. Therefore, we evaluate how our approach meets the following goals:

*G1.* Provide a flexible module-based runtime for software deployment from remote servers or cloud virtual instances to mobile devices – Allowing the seamless distribution of application modules and on-demand code migration in a transparent fashion relative to the user is essential.

*G2.* Reduce the overall interaction time for applications and the power consumption on mobile devices by identifying the optimal distribution of application modules in the mobile – cloud (or remote server) environment – Providing users with shorter and spontaneous interactions, while in the same time extending the battery life of his device, is critical to improving their experience.

*G3.* Dynamic redeployment of application distributions – Our approach needs to provide the ability of switching between distributions, by on-demand and dynamic code migration, without involving the user.

*G4.* Adapt to changes in CPU utilization, network conditions and data loads by adjusting the distribution partitioning on-the-fly – Such variations are likely to be encountered when operating with mobile devices and have a significant impact on the application performance and the device behavior.

*G5.* Maintain a reasonable overhead on the mobile device – With the limited capabilities of mobile devices, our approach needs to maintain a reasonable overhead, that should be preferably comparable to those of any regular applications users interact with on a regular basis.

Finally, we compare the accuracy of the `ALL` and `K-Step` algorithms in choos-


(a) Indoor localization (b) Text-to-speech synthetizer (c) Ticket machine

(d) Interior decoration

**Figure 4.1:** Screenshots of prototype applications on HTC Desire (a–c) and on Nokia N810 (d).

ing the optimal application distribution and discuss what are the scenarios in which these optimization strategies would be most suitable.

## 4.1 Prototype Applications

AlfredO's goal is to allow a user to flexibly interact with cloud-resident applications, as well as physical service infrastructures. We have implemented four prototype applications that provide examples for both classes. Two of our applications belong to the *media* and *maps* categories, which are among the top five most popular categories for mobile devices (FMK$^+$10). The third application for interaction with ticket machines is an example of an infrastructure service, while the last application performs image processing functions to aid users in

the process of interior decorations. Screenshots for these applications are shown in Figure 4.1. All applications have been modularized either from scratch or from existing monolithic versions and follow the practical guidelines formulated in Chapter 3.3.2.

**Interior Decoration Application.** The prototype panorama application implements several image composition functions of an interior design application. 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.

The entire application consists of 19 bundles with varying requirements in terms of processing and communication resources. The service dependencies between bundles generate the graph configuration shown in Figure 4.2, where we can identify two flows of bundles that process the small image of the furniture item through the *mSmallImage* bundle and the large image of the house through the *mLargeImage* bundle separately, and then merge through bundles *mCrystallize* and *mContour*. The heavy computation bundles, namely *mOil*, *mCrystallize* and *mContour*, are marked as non-movable to the mobile device by the developer and are shown as dark-grey in the figure.

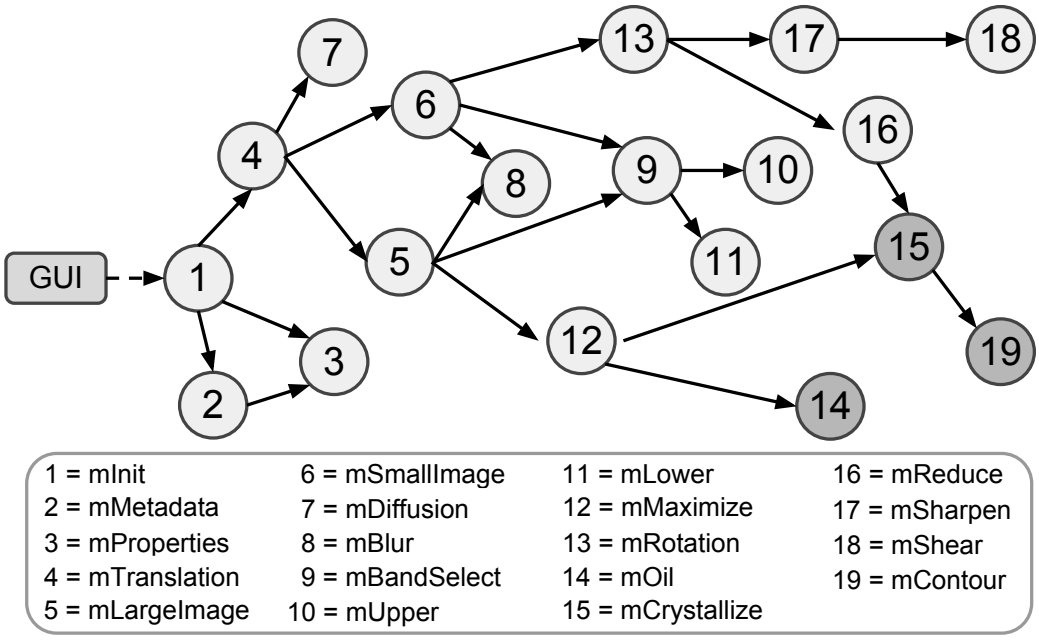



**Figure 4.2:** Modularization of the interior decoration application.

**Ticket Machine Application.** The ticket machine application allows users to

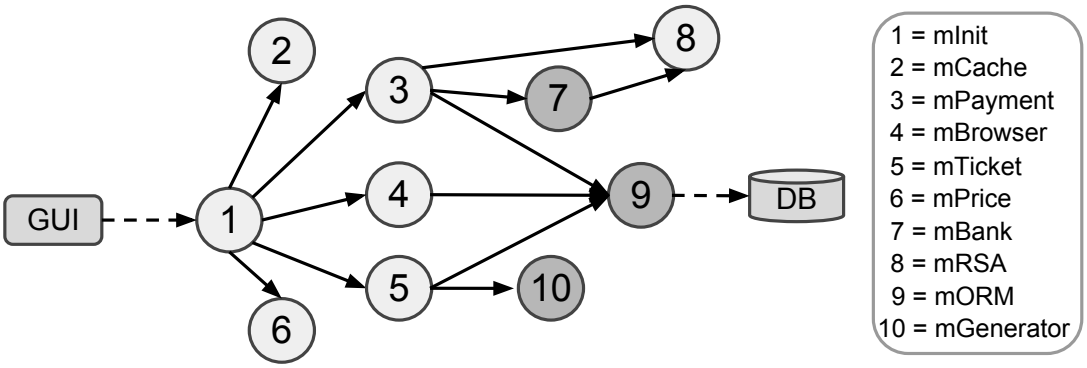



**Figure 4.3:** Modularization of the ticket machine application.

purchase train tickets, browse train routes, check prices and receive electronic tickets from their mobile devices. The motivation for this application came from a joint project with the Swiss national railway (SBB), in order to provide a centralized version of the application for users, while traveling around Switzerland.

As shown in Figure 4.3, the application has been implemented using 10 bundles. *mInit* sets parameters and user preferences. *mORM* executes queries over a database containing train routes, prices, sold tickets and transaction details. The queries are issued by *mRBrowser*, *mPayment*, and *mTicket*. All searched routes as well as customer details can be cached by *mCache*. The overall price for purchasing full- or half-fare tickets is computed by *mPrice*. The application allows for online payment and uses the *mRSA* module to encrypt and decrypt customer details, which are then passed to an external bundle *mBank* that balances the customer's account after the purchase. Finally, the user is presented with an image-generated electronic ticket by *mGenerator*. In Figure 4.3, three modules are marked as *non-movable* to the mobile device: *mORM*, directly connected to the database, *mGenerator* and *mBank*, which both contain private SBB data.

**Indoor Localization Application.** The indoor localization application provides users with visualization facilities of a building map, localization on the map, browsing and directions to different people and places within the building. Indoor localization is carried out using the mobile device's camera. This technique takes inspiration from the application proposed in (RSF+06). A possible scenario is that of a student interviewing for a job in a foreign university. Having an interview schedule, the visitor can easily browse for people to meet, locate himself inside the building, and receive directions on how to reach different offices.

As shown in Figure 4.4, the application's functions have been implemented using 12 bundles. The bundle *mInit* sets parameters and user preferences. *mMaps* and *mBrowser* allow the user to choose buildings, places and people for which maps are retrieved by *mORM* from a database. *mCache* can save the searched

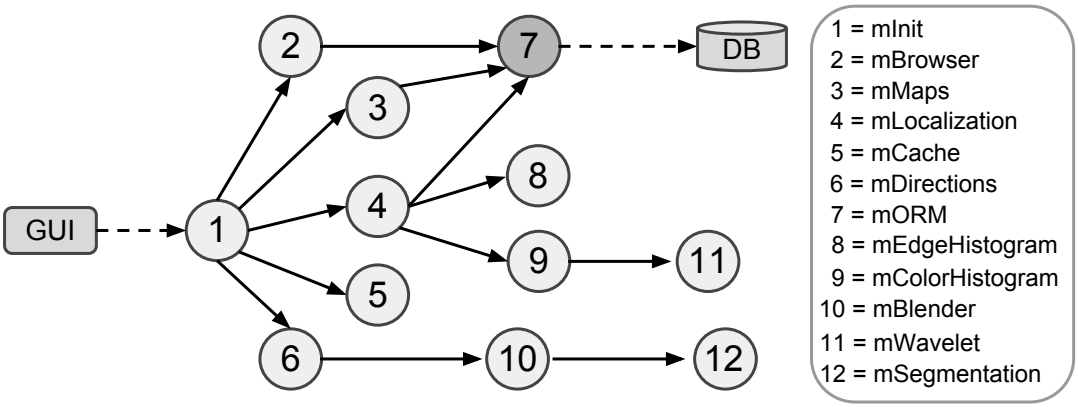



**Figure 4.4:** Modularization of the indoor localization application.

maps for future use. To locate themselves inside a building, users need to take one or more photos of their surroundings, which are then compared for similarity against existing snapshots in the database. To determine the similarity degree, photos taken by the user 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. For this application, only *mORM* is marked as *non-movable*, since it is strongly coupled with the database.

**Text-to-speech Synthesizer Application.** The text-to-speech translation application supports two operations: (a) the translation of a text extracted from a photo taken with the mobile device's camera, and (b) the generation of speech from the translated text. The application has been implemented by adapting modules from the FreeTTS (Fre10) synthesizer.

As shown in Figure 4.5, the application consists of 10 modules. *mInit* sets initialization parameters, *mOCR* performs optical character recognition from an image to extract text, and *mTranslate* translates the resulting text into another language, using the Google Translate API. *mTTS* performs an initial processing of the input text and collects the generated speech as an output stream. The remaining bundles synthesize the translated text into speech: *mEnglish* provides English support for the offered voices and implements English grammar checking. *mCart* implements several classification and regression trees used to make decisions on the duration and structure of phrases. *mLexicon* provides a list of words based on letter-to-sound rules, while *mRelp* implements a linear predictive decoding of audio samples, as well as encoding of diphones. *mAudio* provides generic audio support and *mUtils* offers additional tools and utilities. For this

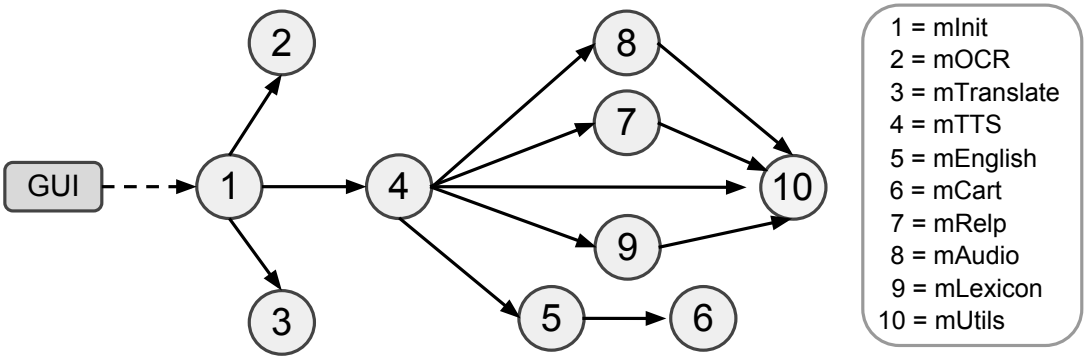



**Figure 4.5:** Modularization of the text-to-speech synthesizer application.

application there are no restrictions on the location of bundles.

## 4.2 Experimental Evaluation

Our evaluation was performed in two different setups. In the first scenario the mobile device runs on a Nokia N810 Internet table and the server on a regular laptop computer with an Intel Core 2 Duo T7800 processor at 2.6GHz. N810 hand-held devices, 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. The N810 device was connected to the laptop either through WiFi (IEEE 802.11b) in ad hoc mode or through bluetooth. For this specific setup, we used the interior decoration application described in Chapter 4.1.

In the second case, our intention was to evaluate how AlfredO performs when the client executes on a HTC Desire smartphone and the server on small, medium or large standard Amazon (EC2) EC2 virtual 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 remote side using WiFi (IEEE 802.11 b/g) or 3G. The small, medium and large EC2 instances have the following specifications: (1.7GB RAM, 1 core, 169GB storage), (3.75GB RAM, 2 cores, 410GB storage) and (7.5GB RAM, 4 cores, 850GB storage), respectively. For all experiments in this particular setup, we used the indoor localization, ticket machine and text-to-speech synthesizer applications described in Chapter 4.1.

### 4.2.1 Initialization Cost

We start by characterizing the performance overhead of AlfredO on the Nokia N810 and HTC Desire mobile devices. The start up process consists of launching the AlfredO client components shown in Figure 3.1 and registering their inter-

**Table 4.1:** Startup time (average and [standard deviation]) with bluetooth for selected configurations of the interior decoration application.

| Configuration (list of bundles) | Fetch & Install + Start size (bytes) | time (ms) | Proxy generation num | time (ms) | Total time (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 |

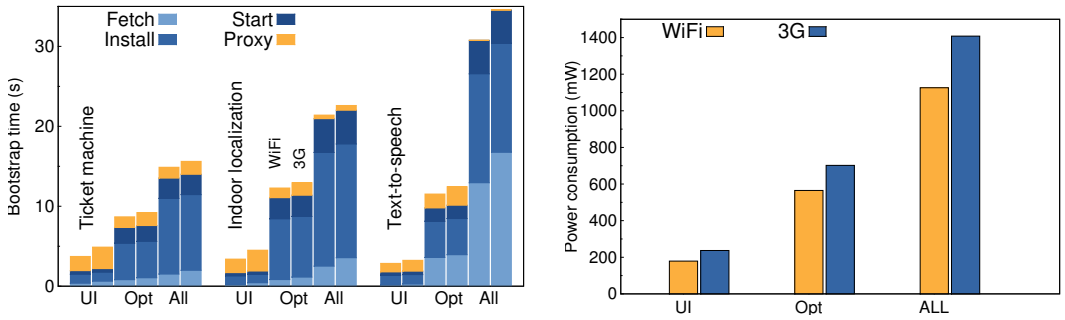
dependencies. This takes on average 12 – 14 seconds on the HTC Desire and 16 seconds on the Nokia N810. Since in idle mode the system consumes few resources, the bootstrap can easily take place when the mobile device is turned on. Once AlfredO is running, the startup time of an application varies depending on the module distribution between the client and the remote server or the cloud virtual instance, respectively. More precisely it depends on the number of application modules that need to be installed and started at both ends, the number of services to be registered with the local registries, as well as the number of proxies that need to be generated for all remote dependencies.

#### 4.2.1.1 Nokia N810 – ThinkPad T61p

In the Nokia N810 – ThinkPad T61p environment, we start by analyzing the startup time of selected configurations for the interior decoration application. We measure the time necessary to fetch, install and start the 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 4.1.

The fetching time obviously increases with the number and size of the bundles acquired. The installation and starting times are typically cumulated to 1 – 1.5 seconds per bundle. The proxy generation time depends on how many service dependencies exist with remote bundles, but in average it takes around 300 – 400 mseconds. For example, in the first case, `mInit` has 3 dependencies, namely `mTranslation`, `mProperties`, and `mMetadata`. Although the fetching and installation overhead can be even 30 seconds, as in the last case when 15 bundles are acquired, the optimization algorithms opt for these kinds of configuration only when the performance gain is high enough. For smaller configurations, as in the first few cases, the overhead is comparable to the startup time of common applications on mobile devices, such as text editors, web browsers or map navigations.

Once the interaction with an application ends, all the modules that have been




(a) Startup time

(b) Power consumption

**Figure 4.6:** Startup time and power consumption on HTC Desire for the ticket machine, indoor localization and text-to-speech applications on EC2 US-East instances (WiFi and 3G used).

fetched on the client are discarded in order to free the allocated memory locations on the mobile device. This guarantees that AlfredO consumes from the limited resources of mobile devices only during the actual interaction of a user with an application.

#### 4.2.1.2 HTC Desire – Amazon EC2 Instances

Figure 4.6(a) shows the installation times for the ticket machine, indoor localization and text-to-speech applications. For each application, we report three pairs of bars. For each pair, the first represents the WiFi measurements and the second one the 3G measurements. The first pair of bars reports the installation time for the `UI` configuration in which only the user interface is fetched on the mobile device. The second pair `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 side. Figure 4.6(b) reports how much power is consumed for the indoor localization 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 from the cloud to the mobile device can be greatly reduced by acquiring only parts of it. For the text-to-speech synthesizer, the gap between the `Opt` and `All` installation times is of almost 18 seconds, which is a considerable overhead for any mobile device. We also notice a significantly less power consumption by 600 – 700 mW 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 actually more complex, and analyze how the resulting application interaction time and power consumption

varies with the chosen partitioning configuration.

A more detailed analysis of Figure 4.6(a) 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 1.7 seconds per bundle. Generating one remote proxy takes around 300 mseconds for WiFi and 420 mseconds for 3G. These values are similar to those obtained when using the Nokia N810 device. For the indoor localization application (shown in Figure 4.4), the `UI` setup acquires one bundle and generates five proxies, while the `Opt` configuration fetches six bundles and generates three proxies. The time required to stop-uninstall a bundle is in the range of 1.6 seconds and a proxy removal requires around 350 – 400 mseconds.

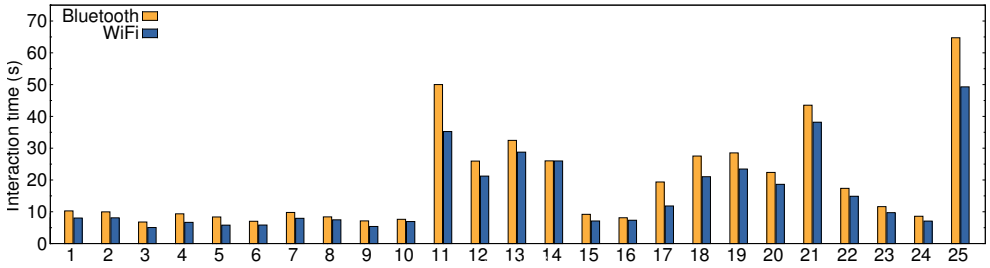
### 4.2.2 Steady-State Behavior

The difference to the previous experiments is given by the fact that we now consider the system's steady state, when all required bundles for a certain partitioning configuration have already been fetched, installed and started on the mobile device. To assess the effectiveness of our algorithms in identifying the configuration that minimizes the overall interaction time of an application, we run several grounding experiments where the performance of the most significant valid distributions of application bundles are quantified. First, we report the results obtained with the interior decoration application, when the setup consists of a Nokia N810 device and a ThinkPad T61p machine. Second, we study the steady-state behavior of the ticket machine, indoor localization and text-to-speech synthesizer applications for an HTC Desire and EC2 virtual instances. For this last set of experiments we also discuss the power consumption effect on the mobile device.

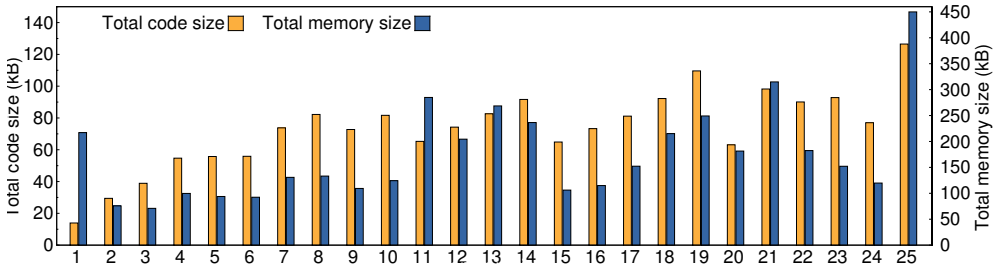
#### 4.2.2.1 Nokia N810 – ThinkPad T61p

Figure 4.7(a) shows the interaction time both with WiFi and bluetooth for the interior decoration application, while Figure 4.7(b) displays 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 $< 100KB, 30KB >$, where the 100KB image corresponds to the house picture and the 30KB image represents the furniture item picture. As the number of configurations for the given application's graph is more than 100, we report results for 25 relevant configurations, such that bundles are gradually migrated to the mobile device (i.e., from configuration 1, when all bundles are running remotely, to configuration 25, when all are executed locally).

These experiments allow us to draw two important conclusions. First, there




(a) Interaction time



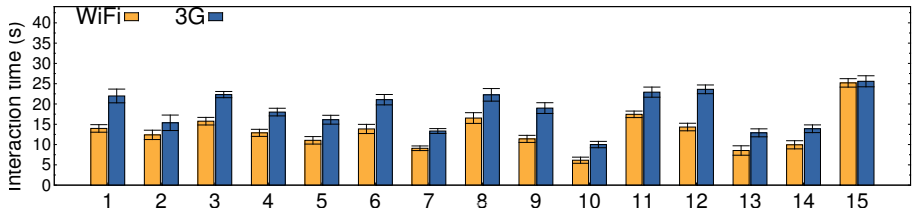

(b) Resource consumption

**Figure 4.7:** Overall interaction time (a) and resource consumption (b) for the interior decoration application (WiFi and bluetooth used).

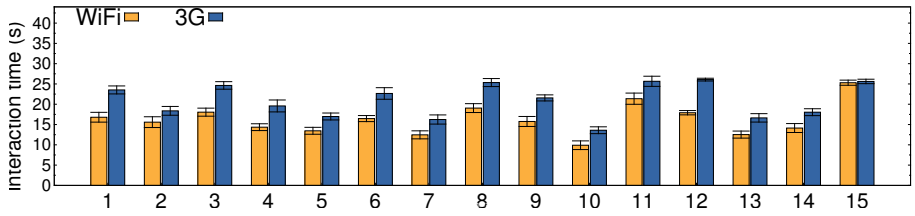
is no clear correlation between the interaction latency observed by the end-user and the number and size of bundles acquired by the client. There are cases in which acquiring more code does increase the interaction latency of more than 10 seconds (for example, passing from configuration 20 to 21) and 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 one configuration to the other. This indicates that there is potential to use the proposed algorithms to select the best configuration.

#### 4.2.2.2 HTC Desire – Amazon EC2 Instances

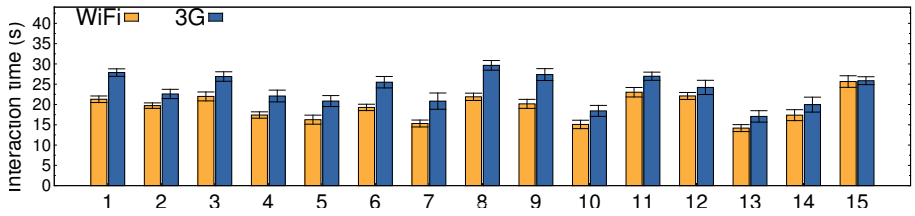
Figures 4.8–4.10 report the user-observed interaction time for a subset of the possible partitioning configurations for the ticket machine, indoor localization and text-to-speech synthesizer applications on three different types of EC2 instances (small, medium and large). Configurations are ordered by increasing number of bundles acquired on the client. Configuration 1 always represents the scenario where only the application user interface is installed on the device (`UI`), while the last configuration corresponds to running the entire application on the client side (`All`). We note here that in the case of the indoor localization and the ticket machine, some bundles are bound to be executed on the virtual instances (i.e.,




(a) Indoor localization (US-Large)




(b) Indoor localization (US-Medium)

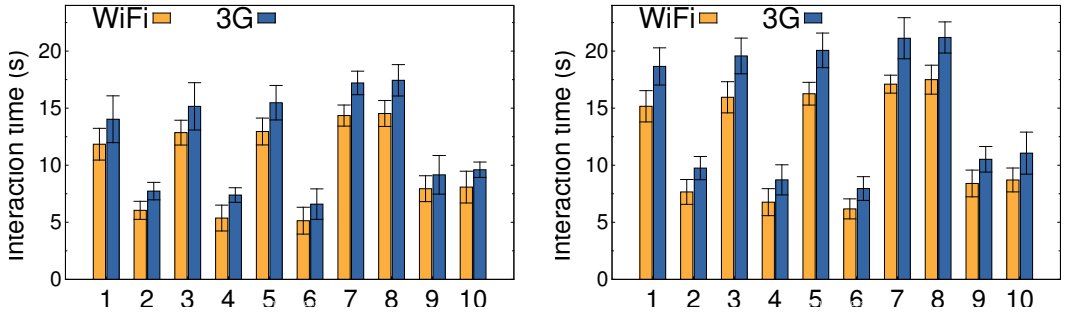



(c) Indoor localization (US-Small)

**Figure 4.8:** Interaction time for indoor localization application with varying configurations, on different EC2 instances (WiFi and 3G used).

`mBank`, `mGenerator` for the ticket machine and `mORM` for both applications).

Acquiring more bundles on the mobile side does not necessarily improve performance, and installing only the application's user interface 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, which are consistent with those obtained for the interior decoration application, motivate the need for a partitioning algorithm capable of choosing the most suitable partition. The optimal partitioning for the indoor localization depends on the EC2 instance type, with configuration 10 (i.e., `mInit`, `mMaps`, `mBrowser`, `mCache`, `mDirections`, `mBlender`) for large and medium instances and 13 (i.e., `mInit`, `mMaps`, `mBrowser`, `mCache`, `mDirections`, `mBlender`, `mEdge`, `mLocalization`) for small machines. Similarly, for the text-to-speech synthesizer configuration 5 (i.e., `mInit`, `mTTS`, `mLexicon`, `mAudio`) is best for large instances and 6 (i.e., `mInit`, `mTTS`, `mLexicon`, `mTranslate`) for medium and small ones. In the case of the ticket machine, the optimal configuration is 6 (i.e., `mInit`, `mBrowser`, `mPrice`, `mCache`) for all instances. Figure 4.11 reports the power con-



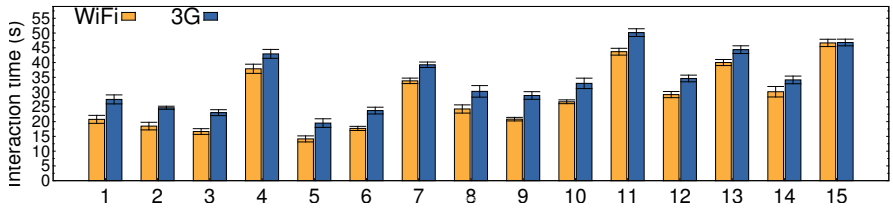

(a) Ticket machine (US-Large) (b) Ticket machine (US-Small)

**Figure 4.9:** Interaction time for the ticket machine application with varying configurations, on different EC2 instances (WiFi and 3G used).

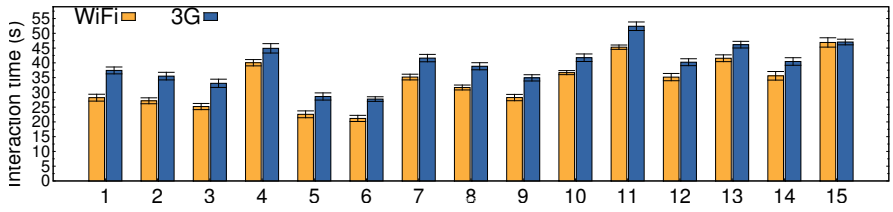
sumed by running the same set of configurations for all applications, but only on large EC2 instances. Our measurements confirm that the optimal configuration in terms of interaction time is also the most power-efficient for all applications, since it consumes the least amount of mW. In addition, we observe similar trends between the power consumption and interaction times for small an medium EC2 instances, where the best configurations for both indoor localization and text-to-speech synthesizer 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 the `ALL` solving algorithm and measure the achieved improvement on the interaction time and power consumption. For all applications, the algorithm is able to select the best configuration. Table 4.2 reports the interaction time of the optimal configuration and the algorithm solving time. The performance gain on the mobile device is very promising. For the ticket machine application the gain is up to 61%, when we compare with the two extreme cases, `All` and `UI`. For the localization and synthesizer applications, improvements are even higher, up to 75% and 69% respectively. Finally, Table 4.2 reports the power consumed by running the optimal configuration and the gains AlfredO achieves when compared to `All` and `UI`. Given the similar trends between power consumption and interaction time described earlier, we notice for all three applications a lower amount of mWs consumed by 20 – 46%, with smaller gains for 3G as expected.

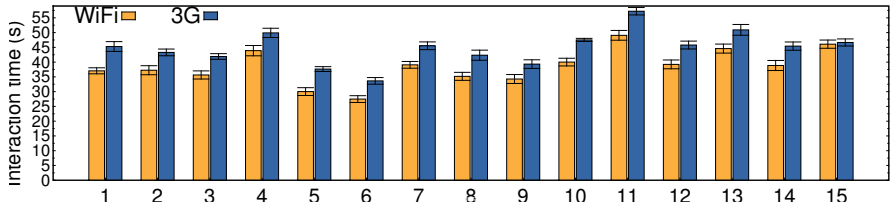
To understand how performance is improved we need to consider the applications' bundle structure. For example, in the case of the indoor localization application shown in Figure 4.4, the best configuration contains `mInit`, `mCache`,




(a) Text-to-speech (US-Large)




(b) Text-to-speech (US-Medium)



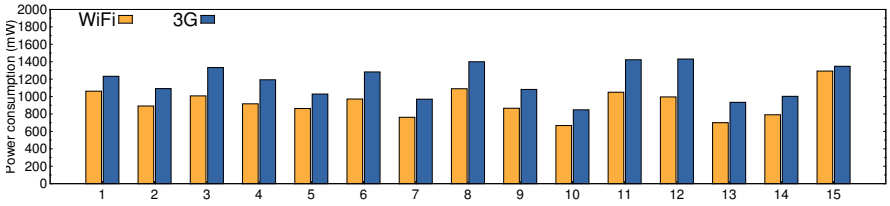

(c) Text-to-speech (US-Small)

**Figure 4.10:** Interaction time for the text-to-speech application with varying configurations, on different EC2 instances (WiFi and 3G used).

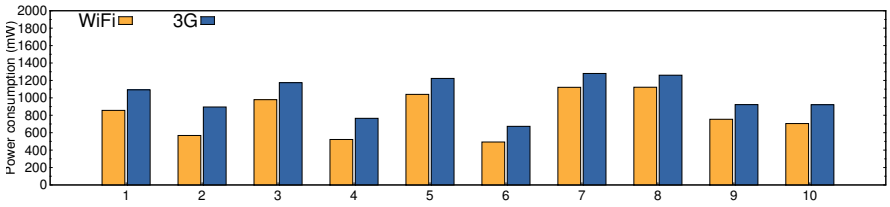
`mMaps`, `mBrowser`, `mDirections` and `mBlender`. For this application, it is not convenient to fetch more bundles because they are too computational intensive for the mobile platform and their execution time on the client exceeds the time required for transferring the required inputs to the cloud.

As explained in the previous sections, once the AlfredO 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 of distributing an application between the cloud and the mobile device can bring such a high performance improvement to fully hide the installation overhead. This makes the approach suitable for both long-term and short-term interactions.

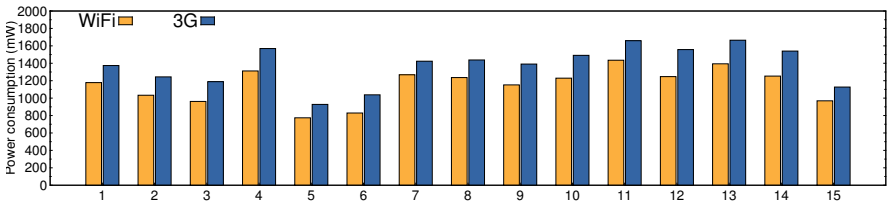
Finally, we look again at the power consumption gains of `Opt` over `All` and `UI`




(a) Indoor localization (US-Large)




(b) Ticket machine (US-Large)




(c) Text-to-speech (US-Large)

**Figure 4.11:** Power consumption for the ticket machine, indoor localization and text-to-speech applications with varying configurations, on EC2 US-Large instances (WiFi and 3G used).

reported in Table 4.2. Although the values obtained already show the advantages of application distribution, for users it is more meaningful to understand how these apply to the battery life on their devices. Therefore, we first consider the specifications of the HTC Desire smartphone used in the experiments and compute the total power available. Since the device's battery is standard Li-Ion at 1400 mAh and 3.7 voltage, the total power is the multiplication of the two, namely 5.18 Wh. In the next step we need to compute the total energy available, by using the previous result and the average number of standby hours. Although the HTC Desire specifications mention up to 300 hours standby, with smartphones that have been used over longer spans of time this is hardly the case. Therefore, we choose a different methodology as described in (TAN$^+$12b). During 600 seconds the device was stressed by running several consecutive interactions of the indoor localization application with the optimal configuration. At the end of the 600 seconds we observed a total energy of 441 Joules that resulted in a 22% drop in battery charge. From this experiment we learned that 1% drop in battery

**Table 4.2:** AlfredO performance and power consumption with the ticket machine, indoor localization and text-to-speech applications on large EC2 instances.

| | | Ticket machine | | Indoor localization | | Text-to-speech | |
|---|---|---|---|---|---|---|---|
| Solver | | 0.16s | | 0.2s | | 0.17s | |
| | | Time (s) | Imp. | Time (s) | Imp. | Time (s) | Imp. |
| OPT | WiFi | 5.17 | | 6.13 | | 14.14 | |
| | 3G | 6.72 | | 10.01 | | 19.05 | |
| ALL | WiFi | 7.79 | 33% | 25.3 | 75% | 46.66 | 69% |
| | 3G | 9.5 | 29% | 25.59 | 60% | 46.8 | 58% |
| UI | WiFi | 11.84 | 57% | 13.97 | 56% | 20.76 | 31% |
| | 3G | 14.03 | 52% | 21.97 | 54% | 27.52 | 29% |
| | | Power (mW) | Imp. | Power (mW) | Imp. | Power (mW) | Imp. |
| OPT | WiFi | 493 | | 668 | | 774 | |
| | 3G | 673 | | 848 | | 928 | |
| ALL | WiFi | 705 | 30% | 1243 | 46% | 969 | 21% |
| | 3G | 921 | 27% | 1348 | 38% | 1127 | 18% |
| UI | WiFi | 856 | 43% | 1062 | 37% | 1178 | 34% |
| | 3G | 1092 | 39% | 1233 | 32% | 1374 | 32% |

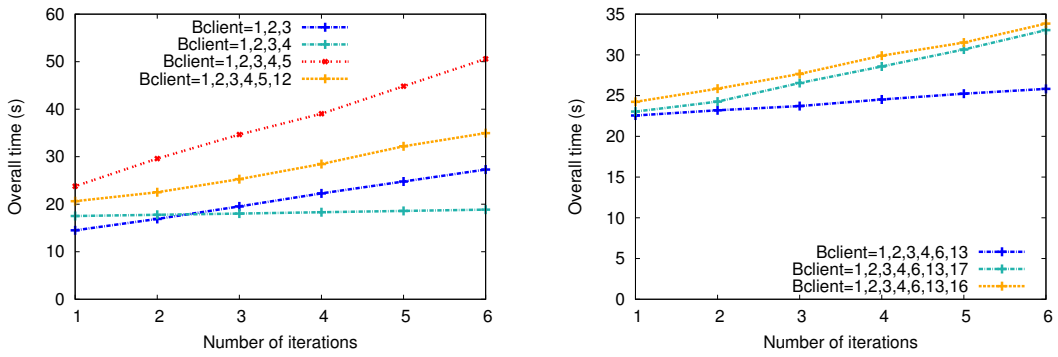
corresponds to a consumption of 20.94 Joules. The last step is converting the measured power consumption into energy and relate it to percentages in battery drop. To achieve this easily, we multiply the power consumption for a particular configuration (`p`) with the duration of its interaction (`t`) to obtain the energy expressed in millijoules. Table 4.3 shows the percentages in battery drop corresponding to the above measured power consumption over 10 user interactions. As expected, the battery drops are higher for the traditional approaches and become more significant for more complex applications, such as the text-to-speech synthesizer.

### 4.2.3 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", which is actually one invocation per every application bundle. However, in reality this rarely occurs. For example, in setting the position, dimension, or rotation of a furniture item in the interior decoration application 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

**Table 4.3:** AlfredO's improvement over battery drop with the ticket machine, indoor localization and text-to-speech applications on large EC2 instances.

| | | Ticket machine | Indoor localization | Text-to-speech |
|---|---|---|---|---|
| | | Battery (%) | Battery (%) | Battery (%) |
| OPT | WiFi | *1.21* | *1.95* | *5.22* |
| | 3G | *2.15* | *4.05* | *8.44* |
| ALL | WiFi | 2.62 | 15.01 | 21.59 |
| | 3G | 4.27 | 16.47 | 25.18 |
| UI | WiFi | 4.84 | 7.08 | 11.67 |
| | 3G | 7.31 | 12.93 | 18.05 |




(a) Increasing invocations of 4 and 12

(b) Increasing invocations of 16 and 17

**Figure 4.12:** Overall time with multiple service invocations with WiFi.

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 for the interior decoration application. In Figure 4.12 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, as well as 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 shown in Figure 4.12(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 in-

creases. 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 this bundle is above 2. Otherwise, the overhead of acquiring bundle 4 is higher than the benefit provided. With the second pair of configurations, acquiring bundle 12 is always more convenient and with 6 iterations the performance gain is more than 14 seconds.

In Figure 4.12(b), we see the opposite behavior. While with one iteration the performance of all configurations 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 behavior 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).

### 4.2.4 Dynamic Optimization and Redeployment

We further investigate which partitioning configuration is optimal at bootstrap, when no bundle has been acquired yet on the mobile device. The correlation between the best configuration in terms of interaction time and best configuration in terms of installation time is not obvious. If we look again at Figure 4.6 and Figure 4.8, we notice, for instance, that configuration 12 provides the lowest interaction time, but its installation cost is higher than the costs of configurations 10 or 3, which provide some of the highest interaction times.

Ideally, in choosing the best partitioning one should consider the number of user interactions a user will make. In the case of one invocation, the configuration with minimal installation time is probably better, while for a high number of invocations, having a lower invocation time is more important. As the number of interactions cannot be easily known a priori, AlfredO makes the decision entirely online and periodically re-evaluates it, as shown in Figure 4.13.

We consider the scenario in which a user performs three consecutive interactions with the indoor localization application. The optimizer processes the objective function every 10 seconds and returns the optimal configuration. At time 0 when no code is acquired yet on the mobile device, the optimizer picks configuration 2 (`c2` in the figure – consists of the `mInit` and `mCache` bundles),

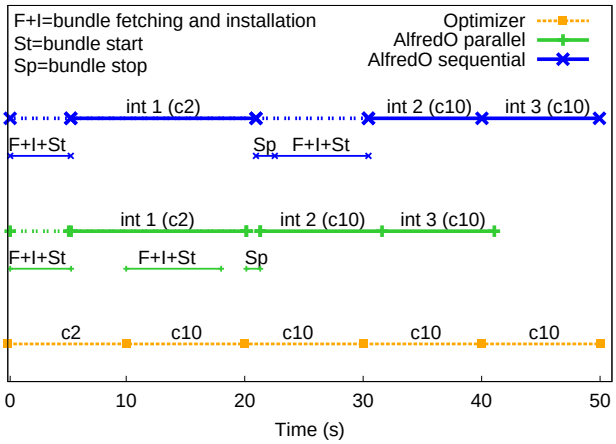


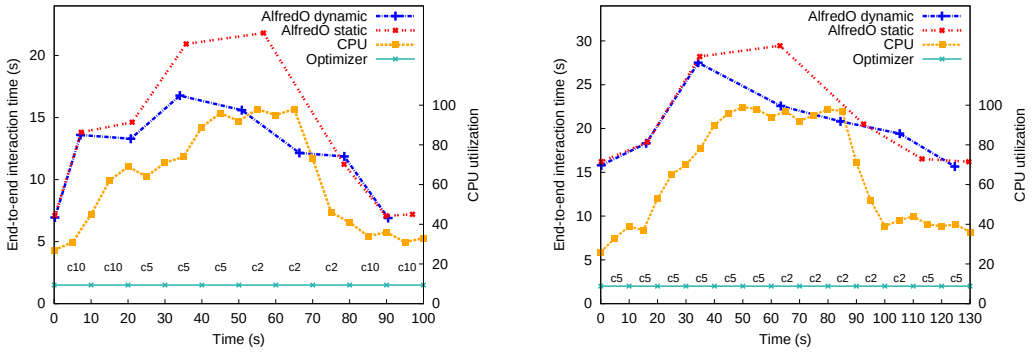

**Figure 4.13:** Parallel bundle initialization for the localization application with 3 consecutive user interactions (3G in use).

which does not provide the lowest interaction time, but minimizes the overall time for initialization and interaction. At the next run, the optimizer re-evaluates the objective function which now yields a much smaller initialization cost as some bundles have already been acquired on the mobile device in parallel with the application execution. This leads the optimizer to pick at time 10 the configuration with the lowest interaction time, `c10`. Its cost is roughly 8 seconds, since c2 is already active. As the initialization ends before the first interaction has completed, this more efficient configuration can be used already at the second invocation.

Figure 4.13 shows the benefits of initializing configuration 10 in parallel to the application execution (`AlfredO parallel`). In this way, before c10 becomes active, there is an interruption of only 1.2 seconds (`Sp` in the figure) due to the removal of the three 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 (`AlfredO sequential`), AlfredO parallel allows the user to carry out 3 full interactions in the time AlfredO sequential completes 2. Moreover, in the sequential approach, the application is not available for a much longer period of time, roughly 10 seconds (represented with a dotted line).

### 4.2.5 Reactivity to CPU and Network Variations

The ability of reconfiguring an application online is useful during the startup phase, but also to quickly react to changing network conditions or CPU load. We give an example of both with the next experiment. First, we cause an increase in the device's CPU utilization by running for a few minutes a CPU-intensive process. This causes the CPU utilization to increase to 67%, then to 95% and finally



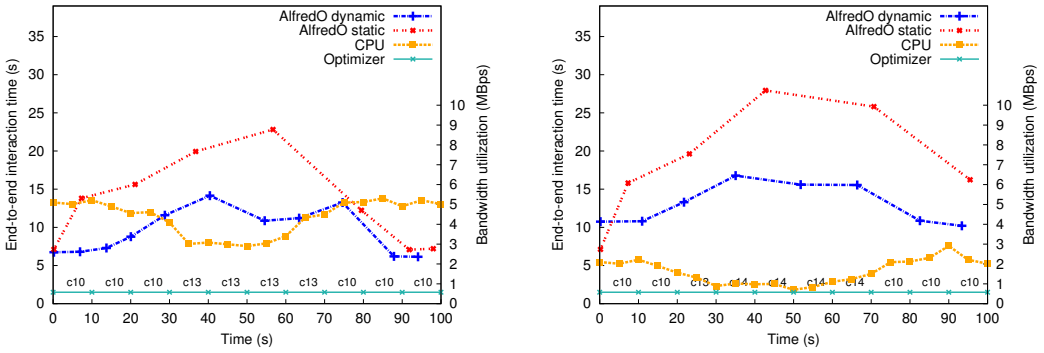

(a) Indoor localization

(b) Text-to-speech translation

**Figure 4.14:** Reactivity to changes in CPU load for the localization and text-to-speech applications (WiFi is used).

to return to normal. The optimizer (executing every 10s) reacts to the CPU variations by choosing a "cheaper" configuration in terms of CPU consumption. Figure 4.14 shows the performance improvement our approach (`AlfredO dynamic`) has over a static one (`AlfredO static`), which always runs with the configuration chosen at the first interaction and does not perform online profiling. The experiment starts after the optimal configuration was installed in parallel with the first interaction.

As shown in Figure 4.14(a), the first interaction with the indoor localization application uses configuration 10. When the first increase in CPU occurs, from 31% to 67%, this configuration provides a longer execution time, by roughly 5.5 seconds. On receiving the new profiled data, the optimizer decides it is better to switch to configuration 5, which contains 4 of the 6 bundles from configuration 10. This decision reduces the CPU requirements on the mobile device, thus providing a performance improvement of 5 seconds over the static approach, which is visible in the fourth interaction. Between the 4th and 5th interactions, an additional increase to 95% occurs. Similarly, the time required to execute configuration 5 increases and the optimizer reacts by choosing configuration 2, which implies reducing the number of bundles running on the mobile device to 2. The performance improvement of AlfredO dynamic compared to the static case is even larger. Finally, we decrease the CPU utilization to 37% and the optimizer decides to switch back to configuration 10. 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 becomes stable. This stability comes from the fact that the optimizer is given the opportunity to periodically verify and strengthen its decision while the appli-



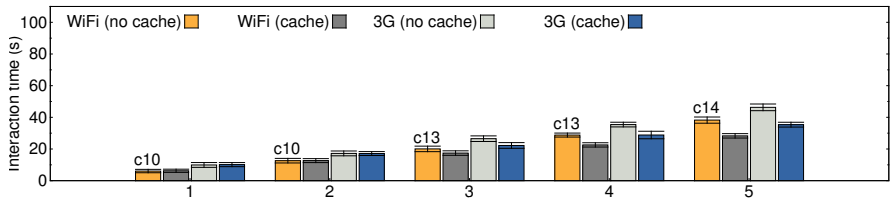

(a) Indoor localization (WiFi)

(b) Indoor localization (3G)

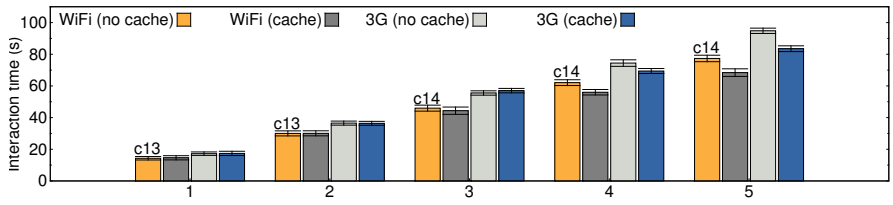
**Figure 4.15:** Reactivity to changes in network bandwidth for the indoor localization application (WiFi and 3G used).

cation is executing. Similar observations hold for the text-to-speech application (Figure 4.14(b)), with the only difference that the optimizer decides to change configuration (from 6 to 2) only after the second CPU increase.

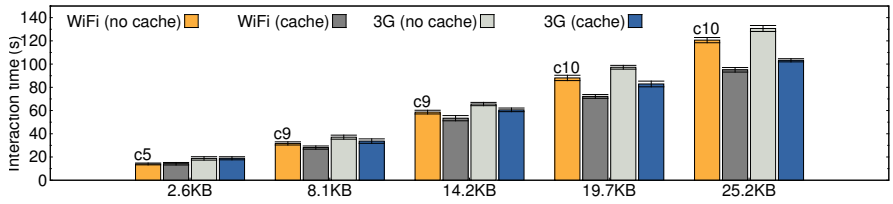
These experiments demonstrate the efficiency of our dynamic approach over a static one. The same mechanism has shown to be effective in reacting to changes in network bandwidth due to unstable wireless connectivity or switching between WiFi and 3G. Figure 4.15 shows how AlfredO adapts the optimal distribution for the indoor localization application for both WiFi and 3G, when drops in the network bandwidth are caused by starting two additional applications that upload and download pictures from a remote site. In the WiFi scenario (Figure 4.15(a)), once the available bandwidth decreases the AlfredO optimizer decides to switch from configuration 10 to 13, which brings more application bundles on the mobile device and reduces the remote data transfer. Once the bandwidth increases, by stopping the background photo downloading and uploading, the optimizer switches back to configuration 10. We notice that with the dynamic approach, a user is able to perform 11 interactions in the same time 8 interactions are executed with the static variant. We apply the same methodology in the 3G scenario (Figure 4.15(b)). Once the bandwidth decreases, the optimizer decides to switch to configuration 13, and immediately after to configuration 14. This means that all application bundles, except for `mWavelet` and `mSegmentation`, are brought on the mobile device. As expected, compared to the WiFi scenario, the lower available bandwidth with 3G connections determines that more computation is executed on the client side. The optimizer switches back to configuration 10, once the background applications are stopped. We observe that the dynamic AlfredO allows users to perform 8 interactions in the same time that 6 interactions would be executed with the static approach.




(a) Indoor localization (US-Large)




(b) Indoor localization (US-Small)




(c) Text-to-speech (US-Large)

**Figure 4.16:** Reactivity to changing user inputs for the indoor localization and text-to-speech applications (WiFi and 3G are used).

### 4.2.6 Adapting to Changing User Inputs

Next, we consider the ability of AlfredO to adapt to changing user inputs. As the optimizer cannot easily predict a user's inputs, the decision of which deployment configuration to adopt is delegated to the client side and is based on a set of usage scenarios and associated distribution schemes cached on the mobile device. We evaluate this feature with the indoor localization and text-to-speech synthesizer applications.

In Figure 4.16, we compare the interaction time for both applications when the cache is disabled or enabled. If disabled, the mobile device adopts the configuration used for the previous interaction. If enabled, the client chooses the best configuration depending on the inputs. For the indoor localization application, we increased the set of inputs from 1 to 5 images, each 300 KB in size. For the text-to-speech application, we varied the size of the text from 2.6 KB to 25.2 KB. Initially, the monolithic version of the application when installed on the mobile device was not able to execute with a text input size larger than 5 KB. This was due to an exponential increase in the interaction time, which would eventually

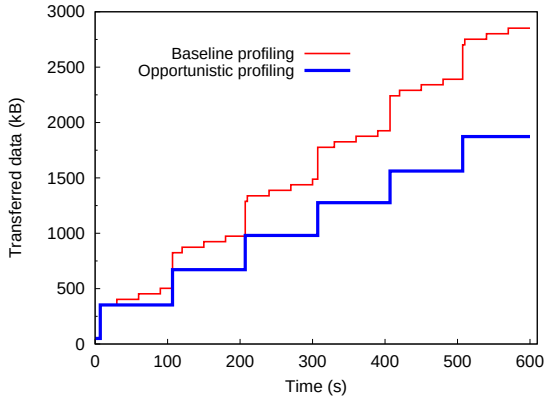



**Figure 4.17:** Opportunistic vs. baseline profiling in a 10-minute scenario with the indoor localization application (WiFi used).

cause the Android operating system to exit the application. With our approach, such applications execute successfully for significantly larger inputs, as shown in this experiment.

On large EC2 instances, both applications show a performance gain of 20-25% when cache is enabled. For the indoor localization, Figure 4.16(a) shows that already with 3 photos it is best to switch from configuration 10 to 13. With 5 or more photos moving to configuration 14 is optimal. In the text-to-speech application (Figure 4.16(c)), AlfredO changes from configuration 5 to 9 at the first increase in text size. With a text increase to 19.7KB, the best configuration becomes 10 and improves performance by over 25 seconds for both WiFi and 3G.

For small EC2 instances, the cache approach decides to switch from configuration 13 to 14 with 3 photos as input (Figure4.16(b)), when in fact configuration 13 gives a lower interaction time for 3G. This is due to the generic nature of the cached solutions, which cannot cover all possible usage scenarios. However, the penalty in performance is very small (1.5 seconds).

### 4.2.7 Resource Overhead

Further we provide more insights on AlfredO's overhead on the mobile platform, and specifically the cost of online profiling and the system's memory footprint. The code size of all AlfredO components residing on the mobile device is 178KB, while on the cloud side is 903KB. The memory footprint of AlfredO is typically between 6 and 7MB, and is comparable to other applications or processes running simultaneously on the Android platform. The values were obtained by running the `adb shell dumpsys meminfo pid` command, which shows both the native and Dalvik heap allocations for a specific process. Other tools, such as `adb shell procrank` have reported similar numbers for the memory consumption.

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

AlfredO reduces the overhead for measuring the available bandwidth through opportunistic profiling. To show this, we consider a 10-minute scenario in which a user walks inside our university building. While walking, he consults the indoor localization application to get directions, by taking a picture of his current location every 100 seconds. In normal conditions, the network profiler sends 50KB of data every 30 seconds to estimate the current network bandwidth. With opportunistic profiling, the bandwidth estimation is based on the data sent between the mobile device and the server during an ongoing interaction. In Figure 4.17, we compare the bandwidth consumption of the baseline and opportunistic profiling and see that the latter reduces it by 34%, from over 2800KB to less than 1900KB.

**Table 4.4:** `ALL` and `K-Step` performance for single and multiple interactions with the interior decoration application.

| Scenario | Algorithm | One iteration app | | | Multiple iteration app | | |
|---|---|---|---|---|---|---|---|
| | | Conf | $O$ | Error | Conf | $O$ | Error |
| 1MB | `ALL` | 1 | 14.45 | 0.03 | 1,4 | 39.37 | 0.02 |
| 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 |
| 10MB | `ALL` | 1,4 | 11.66 | 0.07 | 1,4,6,13 | 38.2 | 0.05 |
| 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 |
| 20–30MB | `ALL` | 1,4 | 11.66 | 0.07 | 1,4,5,12 | 38.07 | 0.06 |
| 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 |
| 20–30MB | `ALL` | 1,4 | 11.66 | 0.07 | 1,4–6,12,13,16 | 37.76 | 0.06 |
| 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 |

### 4.2.8 Algorithms Performance

The last set of tests quantifies the performance of the two proposed algorithms, `ALL` and `K-Step`. We consider several user scenarios for the interior decoration

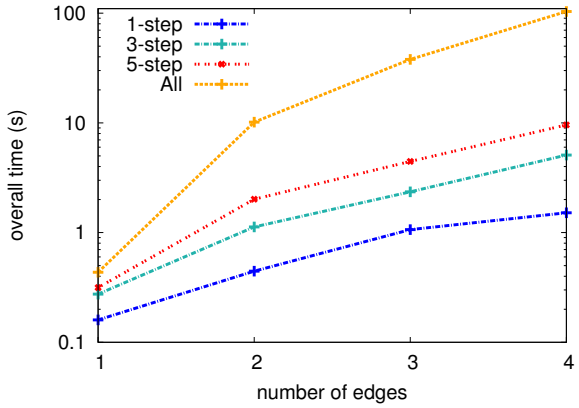



**Figure 4.18:** Processing time for `ALL`, `1-Step`, `3-Step` and `5-Step`.

application, with varying $mem_{MAX}$ and $code_{MAX}$ constraints and two different consumption graphs. Table 4.4 presents the results obtained, where memory and code constraints are in the order of MBs and KBs, respectively.

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 the application bundles multiple 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 mobile device 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 a generic application consisting of 50 bundles and a varying number of bundle dependencies. This results in a consumption graph with 50 vertices and a varying average number of connecting edges. Results are shown in Figure 4.18.

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.

## 4.3 Summary and Discussion

With the four applications considered, AlfredO manages to choose the optimal distribution and improve performance by even 75% when compared to the traditional approaches of either running applications fully on the mobile device or in the cloud. Although AlfredO's interaction latency-based model does not explicitly consider power, we notice that the best performance partitionings are also the most power-efficient and battery consumption on the mobile device is reduced by even 45%. Therefore, we conclude that since network operations consume most power, solving the partitioning problem with the goal of minimizing overall data transfers also enables battery savings.

Experimental results also show that changing the type of virtual instance produces not only variations in the overall interaction time (i.e., expected given the higher or lower computational capabilities of these instances), but also in the optimal distributions. Thus, with small EC2 instances, the best partitionings might imply acquiring and executing more application bundles on the mobile device, if the data transfer cost cannot be amortized by the computational speedup on these instances. A similar effect can be observed when adapting to 3G variations. This is due to both a decreased bandwidth availability and a significantly higher latency, when compared to WiFi. In adapting to user inputs, a simple approach like AlfredO's caching strategy can already provide significant gains in the interaction latency experienced by the user. By employing learning techniques, such as regression-based, these improvements could be increased even further.

Finally, our evaluation shows that AlfredO has a reasonable overhead on the mobile platform and its memory consumption is comparable to any application running nowadays on such devices. Moreover, we see that online profiling with structural reflection incurs a relatively small penalty in performance and a negligible cost in storage space. The comparison between the `ALL` and `K-Step` algorithms shows the ratio between accuracy and cost, with `ALL` always finding the optimal distribution, but at a high cost (i.e., especially as the number of application bundles and their dependencies increases). On the other hand, `K-Step` is significantly faster, although less accurate, which makes it suitable for dynamic adaptations, that are less sensitive to sub-optimal partitionings.

# Chapter 5

# Performance Estimation for Mobile-Cloud Applications

In the context of integrating cloud-based applications with mobile devices, we proposed a flexible technique that optimally distributes these applications to provide users with optimal interaction times and extended battery life. However, our assumption so far was that factors such as workloads or the resource capabilities of the distribution setup are known in advance and profiling data is already gathered through previous runs. In practice, given that these factors are increasing in variety, a-priori knowledge on the application behavior cannot be made available for all their possible combinations. Therefore, to alleviate this problem, in this chapter we propose a performance model that is able to understand how interaction times are impacted by the application behavior, inputs and environment factors. By doing so, it is able to accurately determine how to optimally distribute an application, without executing it in advance and without the high cost of extensive measurements.

## 5.1 Motivation

Until recently, enabling cloud-based applications on mobile devices encompassed two traditional approaches. Either the application is installed on the mobile platform, or the computation remains in the cloud and only the user interface is present on the device. However, both techniques prove unsuitable either due to the computational limitations of devices or the variations in network bandwidth and latency. Therefore, their lack of flexibility cannot provide users with reduced interaction times and battery savings. Our approach, implemented in AlfredO, alleviates these problems by proposing a flexible model that optimally and dynamically distributes applications between the mobile device and the cloud.

While several example applications, either built from scratch or already ex-

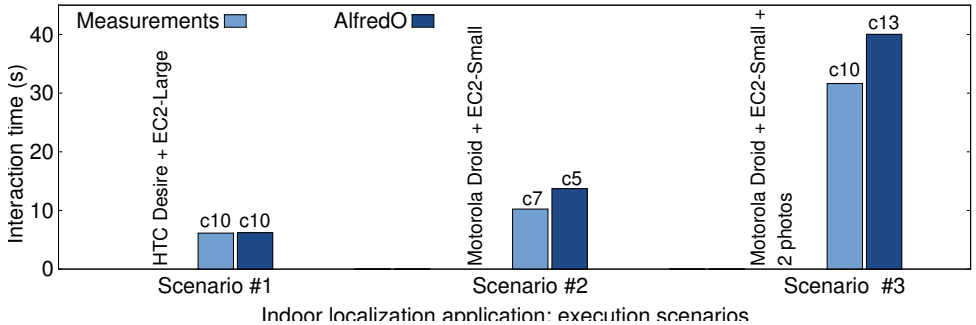



**Figure 5.1:** Comparison between the measured optimal distribution and the configuration chosen by AlfredO in various execution scenarios of the indoor localization application when extensive statistics are not available (WiFi used).

isting, showcase AlfredO's ability to choose the application partitioning with the lowest interaction time even under varying factors, the cost of online profiling is non-negligible. AlfredO requires profiled statistics of the application modules execution times, their communication model and the amount of data transferred between dependent modules during an interaction. This kind of knowledge is easily obtainable by running the application within a time window in a static environment, where the specific mobile platform and cloud instances used for execution are known in advance. However, the application behavior changes whenever different mobile devices or virtual instances are used for deployment, a situation bound to frequently occur in practice. In such scenarios, AlfredO would not be able to optimally distribute applications based solely on statistics gathered for static environments. We present such a scenario for the indoor localization application described in Chapter 4.1 in Figure 5.1. Statistics are gathered for the scenario where the application is distributed between an HTC Desire device and an Amazon EC2 large virtual instance.

When both the mobile device and the cloud instance are replaced by a Motorola Droid smartphone and an EC2 small instance, respectively, the previously collected measurements cause the optimizer to choose a suboptimal partitioning. In this scenario the user experiences a penalty in interaction time of around 4 seconds. Further, when the user inputs two photos, the difference between the optimal partitioning and the one chosen by AlfredO's optimizer is over 9 seconds. These scenarios show that optimal decisions require the application behavior to be monitored for each distinct combination of distribution schemes, workloads, mobile platforms and virtual instance types. However, the space obtained by combining all these factors is extremely large and shows that gathering statistics for all possible scenarios is not tractable in practice. For example, most of the example applications we consider have hundreds of possible distributions. This factor correlated with cloud virtual instance types and various mobile devices with

different sets of capabilities generates possible such combinations in the order of tends of thousands or more.

In this sense, we tackle the complexity problem of online profiling by proposing a performance estimation model (Giu12), built in the context of AlfredO. A key feature of the model is that, while using only passive statistics, it addresses the question: What is the optimal (or nearly optimal) performance of an application with specific workloads, when distributed between certain mobile devices and cloud instances? Performance estimation models have been extensively researched and proposed to analytically model distributed multi-tier Web services (UPS+05, UPS+07, SS05), workload mixes (BDIM04, MAFM99, SKZ07), popular applications (e.g., Mozilla, Visual Studio or Microfost Office) (TDZN10), virtualized resource allocations (WMG+10) or requirements of mobile device tasks (SPPM11). Various approaches to build such models have been considered, from queuing networks to Petri nets, probabilistic formulations or regression-based methods. To the best of our knowledge, the proposed model represents the first attempt to estimate the performance of modular applications, distributed in heterogeneous environments composed of mobile devices and cloud virtual instances. Given the nature of modular applications, we argue that queuing networks naturally emulate their structure and behavior, thus our model is queue-based. The effort of building such a performance model for mobile-cloud applications results in three main conceptual challenges that we discuss further on: (a) identifying the relevant factors that impact performance, (b) unifying all impacting factors under the same model, and (c) bounding factors variability to reduce model complexity, while maintaining good estimation accuracy.

### 5.1.1 Integration in AlfredO

The performance estimation model is built as a module in the AlfredO system, as shown in Figure 5.2. Given that the model requires previously gathered statistics to estimate the performance of applications in specific mobile-cloud environments, we leverage AlfredO's capabilities to provide such knowledge. The application profiler outputs the application graph structure, its module execution times and the amount of data transferred between communicating modules. The device profiler outputs the resource capabilities of the mobile device, its CPU usage, the network type currently in use, as well as its bandwidth and latency. The model provides the abstract representation of applications as queuing networks, as well as the algorithms used to estimate the applications performance based on previous measurements.

By integrating the model in AlfredO, it becomes an alternative to the optimizer in specific scenarios. We remember that the optimizer requires accurate statistics about the application behavior and the device status in order to choose the optimal distribution. In situations where the current execution setup differs

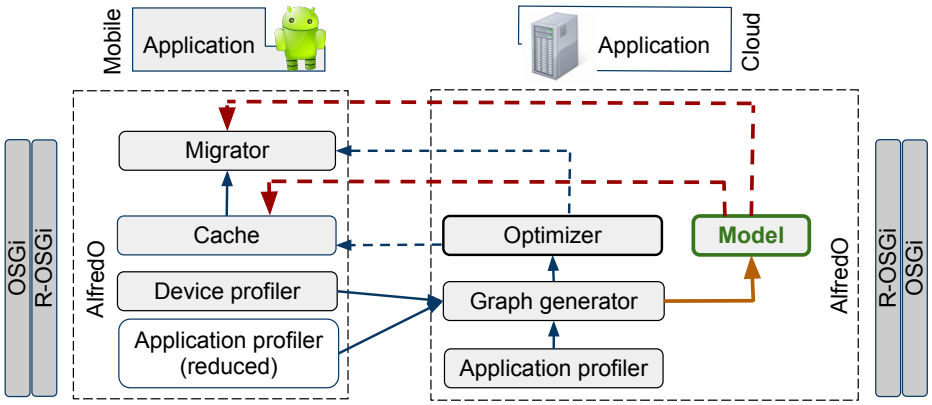



**Figure 5.2:** Integration of the performance estimation model with AlfredO.

from all logged ones for which statistics have been gathered, the estimation model replaces the optimizer in making a partitioning decision. Once such an estimated distribution is chosen, the migrator receives the list of application modules to migrate from the cloud and install on the mobile device. Similarly, the cache keeps a list of pre-computed partitionings with the model that allow the device to autonomously adjust the application deployment relative to user inputs.

## 5.2 Identifying Impacting Factors

The first step in modeling modular mobile-cloud applications is to identify the most relevant factors that highly impact their performance. Given the high variety of applications users interact with, looking at their *demand for underlying resources and workload* is necessary. To validate this claim, consider the ticket machine example described in Chapter 4.1. This application does not have high computational requirements, but is characterized by frequent, short one-type-of-workload interactions, which are directly impacted by the network latency. On the other hand, an image processing application transfers various amounts of data when interactions are initialized and is computationally expensive, thus demanding both network bandwidth and compute resources.

As already shown in Chapter 4, applications can be flexibly partitioned according to various distribution schemes, and each of these partitionings exhibits significantly different user interaction times. Therefore, a third important factor is the *application distribution schemes*.

Finally, in addition to application-specific factors, the setup in which users interact with applications is relevant. Figure 5.1 shows how varying the mobile device and cloud instance types can lead to different interaction times, and consequently to different optimal distribution partitionings. Understanding the

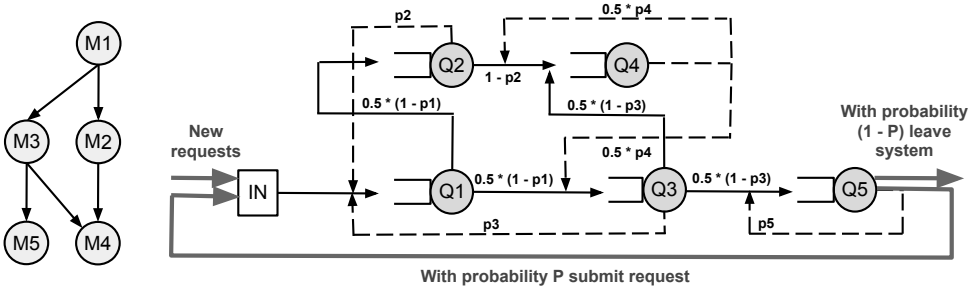



**Figure 5.3:** Modeling an example modular application with queuing networks.

*resource capabilities and usages* of the mobile device, as well as the *capabilities* provided by cloud instances can lead to more informed partitioning decisions, and therefore better user experience. We note that an additional factor is given by investigating the resource usages of the physical machines on which cloud instances are deployed. However, since this kind of information is not made publicly available by cloud providers, we do not include it in the performance model.

## 5.3 Modeling Mobile-Cloud Applications

Consider a modular applications represented as $G = \{\mathbb{M},\mathbb{E}\}$, where the collection of modules is defined as $\mathbb{M} = \{M_i | i = 1, 2, ..., m\}$ and the set of dependencies between $M_i$ and $M_j$ as $\mathbb{E} = \{e_{ij} | i, j = 1, 2, ..., n, i \neq j\}$, as shown in Figure 5.3. $M_1$ represents the entry point in the application and the minimal code that needs to be installed on the mobile device to start interacting with the application. Typically, $M_1$ can implement the application user interface, while the remaining modules implement main logical functionalities. The communication between modules is modeled as a directed acyclic graph, where a module $M_i$ issues requests to all the modules $M_j$ it logically depends on, $i \neq j$, until it reaches the last module $M_m$. In the simplest case, each such request is processed exactly once and the result is sent in reverse order until it reaches $M_1$, which then returns it to the client. More complex processing is possible when a request can visit a module multiple times. As an example, consider a keyword search which triggers queries on different catalogs. In the sequential case, each request is issued once the processing of the previous request has finished. The parallel case is not addressed in this chapter.

We define a partitioning between the mobile device $MD$ and the virtual instance $VI$ as two non-overlapping sets of modules that cover all modules of the application, $P_{MD} = \{M_p | p \in [1, ..., k]\}$ and $P_{VI} = \{M_s | s \in [1, ..., l]\}$, respectively. $P_{MD}$ represents the set of application modules to be migrated on the mobile device, while $P_{VI}$ is the set of modules residing in the cloud. Therefore, $P_{MD} \cup P_{VI} = \mathbb{M}$ and $P_{MD} \cap P_{VI} = \emptyset$. In (GRJ+09, GRA12) we investigate how dif-

ferent distribution partitionings impact application performance, especially when user inputs, device CPU load and network conditions change over time. This confirms the need to understand the performance modeling problem for mobile-cloud applications.

### 5.3.1 Application Resource Demands as Queuing Networks

To understand how resource demanding a modular mobile-cloud application is, we represent it as a network of $m$ M/M/1 queues $Q_1$, ..., $Q_m$. Each queue represents a specific application module and the underlying platform it runs on. When a request arrives at module $M_i$, it triggers one or more requests to modules $M_j$ it depends on; recall the example of a keyword search that triggers multiple queries at different product catalogs. In the proposed queuing model, we capture this by allowing a request to make multiple visits to a queue during its overall execution, and by introducing a transition from each queue to its predecessor. Figure 5.3 represents the queue model for the example application. After processing at queue $Q_j$, a request follows one of two paths based on the application communication model. Either it returns to one of the $b$ queues from which it received the request, $Q_i$, with probability $\frac{p_j}{b}$, or it proceeds to one of the $c$ queues to which it depends on, $Q_k$, with probability $\frac{1-p_i}{c}$.

Let $S_i$ denote the service time of a request at $Q_i$, $1 \leq i \leq m$, and $W_i$ the waiting time to $Q_i$ from all $b$ queues that depend on it. By using these basic queue properties and logged measurements from previous runs, we can model and identify the application demands for specific underlying resources. The service time at $Q_i$ represents the execution time of module $M_i$, which is a measure of the CPU demand per module. The waiting time to $Q_i$ is equivalent with the time required to transfer data from the dependent queues, which is an expression of the network demand (i.e., function of amount of kBs to send, channel bandwidth and latency). By comparing the total service time with the total waiting time, we can understand how much of the total response time is spent performing computational tasks and transferring data, respectively. In the queuing network theory the service time at input, denoted usually by $Z$, represents the user think time. Defining the user think time is particularly useful when there are several clients concurrently interacting with the application, each one issuing multiple sequential requests. In our model we make the assumption that only one user interacts with an application at a moment in time. Extending the model to support more users implies choosing to optimize performance for either every single client or multiple clients. The first scenario is not suitable in practice, because the arrival rate and waiting times of subsequent users impact the performance of the current user, such that the chosen distribution scheme may not be optimal anymore at execution time. In the second scenario, optimizing for multiple users assumes that their number and arrival distribution is known. Even so, the

problem complexity comes from the specific heterogeneous environment for which the estimation model is applied. More precisely, the model should consider all combinations of distribution schemes for all users and estimate which of these combinations globally improves performance. Clearly, the generated combinations space explodes and simplifying solutions are crucial. We leave this aspect as part of future work.

#### 5.3.1.1 Closed, Open and Hybrid Systems

Next, we define how requests are generated under closed, open and hybrid system models, in order to understand which is more suitable for estimating the performance of our target applications. We start with a discussion in the context of multiple concurrent users and conclude whether the same model is suitable for single user scenarios.

In a closed queue system, it is assumed that there exists a fixed number $N$ of users, who use the system indefinitely. Each of these users repeatedly submit requests, receive responses and "think" for a specific amount of time. A new request is only triggered by the completion of a previous request. At all times there are $N_{think}$ users that are thinking and the remaining number of users, $N_{system}$, are either running or queued to run in the system, such that $N_{think} + N_{system} = N$. The response time is defined as the time from when a request is submitted until it is received by the user.

In an open system there is a stream of arriving users with average arrival rate $\lambda$. Each user is assumed to submit one request to the system, wait to receive the response and then leave. The number of users queued or running at the system at any time may range from zero to infinity. The differentiating feature of an open system is that a request completion does not trigger a new request, and a new request is only triggered by a new user arrival. As before, the response time is defined as the time from when a request is submitted until it is completed.

Neither the open nor the closed system models are entirely realistic in practice. Consider for example a ticket machine application, as described in Chapter 4.1. On the one hand, a user is able to make more than one request and will typically wait for the output of the first request before issuing the next one. In these ways a closed model makes sense. On the other hand, the number of users varies over time and setting a fixed number of users $N$ is not justifiable. Therefore, the users visit the ticket machine application and behave as if they are in a closed system for a short period of time, and then leave the system.

Motivated by the ticket machine example, in multiple user scenarios it is important to consider a more realistic alternative, namely a *hybrid* or *partly-open* system, as shown in Figure 5.3. Under the hybrid model, users arrive according to an outside arrival process as in an open system. However, every time a user completes a request in the system, with probability $P$ the user issues a follow-

**Algorithm 3** Baseline MVA for an application composed of $m$ modules.

**Input:** $\mathtt{S}_i$, $\mathtt{W}_i$, $1 \leq \mathrm{i} \leq \mathrm{m}$
**Output:** $\mathtt{R}_i$, $\mathtt{R}$

1: **for** $i = 1 \rightarrow m$ **do**
2: $\quad \mathtt{L}_i \leftarrow 0$
3: $\quad \mathtt{D}_i \leftarrow \mathtt{V}_i(\mathtt{S}_i + \mathtt{W}_i)$
4: $\quad \mathtt{R}_i \leftarrow \mathtt{D}_i(1 + \mathtt{L}_i)$
5: **end for**
6: $\mathtt{R} \leftarrow \sum_{i=1}^{m} \mathtt{R}_i$

| Term | Description |
|---|---|
| $Q_i$ | Queue representing module $M_i$ ($1 \leq \mathrm{i} \leq \mathrm{m}$) |
| $S_i$ | Average service time per request at $Q_i$ |
| $W_i$ | Average waiting time per request at $Q_i$ |
| $L_i$ | Average length of $Q_i$ |
| $R_i$ | Average delay per request at $Q_i$ |
| $R$ | Average response time per request |
| $D_i$ | Average service demand per request at $Q_i$ |
| $V_i$ | Visit ratio for $Q_i$ |

up request possibly after some think time, and with probability $1 - P$ the user leaves the system. Thus the expected number of requests that a user issues to the system during a visit is geometrically distributed with mean $\frac{1}{1-P}$. For a given system load, when $P$ is small, the hybrid model resembles more an open model, while for a large value of $P$, the hybrid model is more similar to the closed model. In (SWHB06), the authors identify important principles of partly-open system models. First, a hybrid system behaves similarly to an open system when the expected number of requests during a user visit is small and similarly to a closed system when the expected number of requests is large. Second, in a partly-open system think time has little effect on the mean response time, and thus can be ignored. By correlating these principles and the behavior of a hybrid system with the variety of applications (i.e., where users can either issue one or more requests per visit), we conclude that these systems represent the most suitable alternative to model such applications.

#### 5.3.1.2 Baseline Derivation

To build the baseline derivation that computes the estimated mean response time experienced by an issued request, we use the Mean-Value Analysis (MVA) algorithm (RL80). The algorithm for an application composed of $M$ modules is presented in Algorithm 3, with the associated notation on the right-hand side.

The algorithm uses the notion of *visit ratio* for each of the $m$ queues, $Q_1$, ..., $Q_m$. The visit ratio $V_i$ corresponding to the queue $Q_i$ represents the average number of visits made by a request-reply interaction to $Q_i$ during its processing, namely from when it is issued by the user up to the point when it returns the result. Visit ratios are easy to compute from the transition probabilities $p_1$, ..., $p_m$ and provide an alternative representation of the queuing network. The use of visit ratios instead of transition probabilities means the model is able to capture

multiple visits to a specific application module regardless of whether they occur sequentially or in parallel. The reason is that visit ratios only express the mean number of visits made by a request to a queue and capture no information about when or in what order visits occur.

In the baseline derivation, we assume that for all requests the waiting times at each application module are equal. Thus, we do not consider how the workload and network variations impact the amounts of data to be transferred between bundles and the transfer times, respectively. The waiting time $W_i$ is expressed as a function of four main parameters (`kB of data, network bandwidth, network latency, connection type`), where the first three remain unchanged. The `connection type` takes one of two possible values, "0" for local connections and "1" for remote connections. With this representation we ensure that any communication cost between two collocated application modules is estimated to 0, while remote communications strictly depend on the amount of data to be transferred and the network properties.

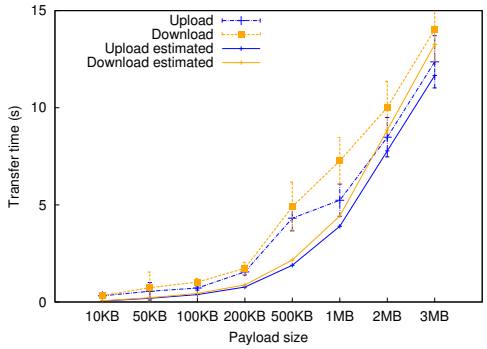
Therefore, given the average service and waiting times, as well as the queues visit ratios, the algorithm computes the average response time $R$ of a request. We note that the $L_i$ parameter, defined as the number of waiting requests at $Q_i$, is an expression of concurrent users. Given that our model only assumes one user at a specific moment in time, we set $L_i$ to 0.

In order to compute the response time, the model requires several input parameters. In practice, these parameters can be estimated by monitoring the application during multiple interactions. To do so, either the underlying operating system or the application provide monitoring hooks that enable accurate estimation of these parameters, or such functionality can be easily implemented. In our scenario, the AlfredO system already provides the necessary infrastructure to gather these required statistics, such as the execution times of application bundles or data transfer times. However, parameters such as visit ratios need to be further estimated. In the remaining of this section, we discuss how the various model parameters are estimated in practice.

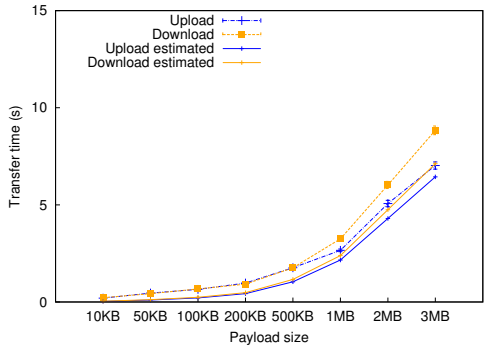
***Estimating visit ratios***. The visit ratio of an application module is the average number of times that module is visited during the lifetime of an interaction. Let $\gamma_{req}$ denote the total number of interactions submitted to an application over a period of time $T$. Therefore, the overall visit ratio $V_i$ for module $M_i$ and the visit ratio $V_{ij}$ for the same module from $M_j$ can be estimated by:

$$V_i \approx \frac{\gamma_i}{\gamma_{req}} \qquad V_{ij} \approx \frac{\gamma_{ij}}{\gamma_{req}} \tag{5.1}$$

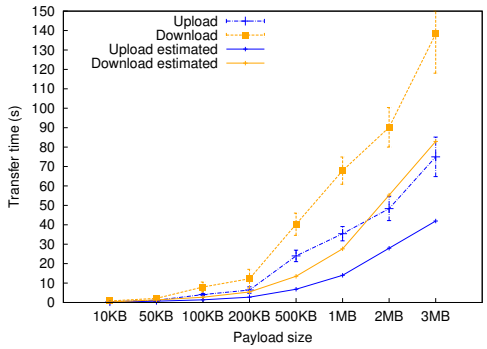
where $\gamma_i$ represents the number of visits module $M_i$ receives, while $\gamma_{ij}$ is the number of visits $M_i$ receives from $M_j$ over time $T$. It is obvious that choosing a small value for $T$ decreases the accuracy of the estimation, since fewer application interactions are considered. We note that visit ratios can be easily estimated



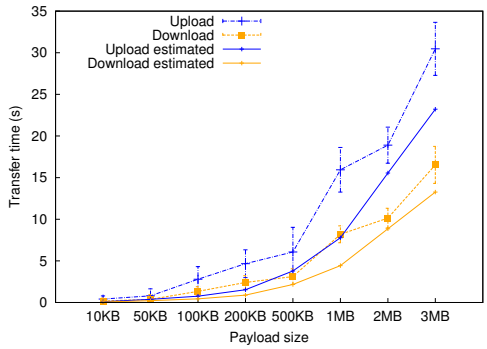

(a) Upload≈2400Kbps, download≈2100Kbps




(b) Upload≈4500Kbps, download≈4000Kbps




(c) Upload≈609Kbps, download≈300Kbps




(d) Upload≈1100Kbps, download≈2000Kbps

**Figure 5.4:** WiFi (a–b) and 3G (c–d) measured and estimated transfer times for Amazon EC2 small and large instances.

through an online approach. The number of requests inputed to an application $\gamma_{req}$ are accounted for at the application entry $IN$, while the requests that reach module $M_i$ are counted by processing logged statistics.

***Estimating waiting times through ground measurements.*** In AlfredO, the network profiling determines periodically what is the upload and download speed on the active connection. Based on this network model, we are able to estimate with relatively high accuracy the download and upload times for different data sizes, as follows:

$$Time_x = \frac{payload}{x_{speed}} + \frac{RTT}{2} + \frac{RTT * payload}{2 * TCP_{window}} \tag{5.2}$$

where $Time_x$ represents the estimated download or upload time for transferring data with size given by the payload parameter. With TCP connections, our model accounts for the initial connection latency as $\frac{RTT}{2}$ and the window size

$TCP_{window}$, which determines how many packets are sequentially sent to transfer the entire payload. We have already shown in Chapter 3.3.5.2 that the default TCP window size is 64KB for both download and upload transfers on mobile devices, while virtual instances usually provide the same window size of 128KB. Thus, given that the bottleneck is the device and the model accounts for the 64KB window size.

Figures 5.4 compares the measured WiFi and 3G transfer times with the estimated transfer times, for various data sizes (i.e., 10KB, 50KB, 100KB, 200KB, 500KB, 1MB, 2MB and 3MB), as resulted from applying Equation 5.2. For WiFi the model follows quite closely the trend of the measurements, with the exception of the scenarios where 500KB and 1MB are transferred to small EC2 instances. The difference between the measured times and the estimated ones for these cases is between 2 and 3 seconds, with an estimation error of up to 56%, due to variations in the available WiFi bandwidth. The same applies to 3G, where increasing the amount of transferred data causes an increase in the estimation error of at most 26% for large instances and at most 45% for small instances. However, even if compared to WiFi the percentage error of 3G is smaller, the gap between the measured and estimated transfer times is larger, with 8 seconds for the large instance scenario for example. To improve the estimation, the current model uses AlfredO's profiler to periodically measure the download and upload speeds, as well as the connection latency, and continuously updates the ground parameters in Equation 5.2 ($x_{speed}$ and $RTT$) with fresh values.

### 5.3.2 Variations for Mobile Devices and Virtual Instances

So far, we have shown how applications can be modeled with queuing networks and how their resource demands for the underlying resources can be identified. Next, it is important to understand how an execution setup, for which the estimation model is applied, influences the interaction time. Thus, we want to quantify how different in terms of available resources is the candidate setup relative to logged setups, for which statistics have been gathered from previous runs.

Let us denote the candidate setup $STP_{current} = (MD_c, VI_c)$ and the logged setup $STP_{log} = (MD_{log}, VI_{log})$. For simplicity reasons, at this stage we only consider a single logged setup. In comparing $STP_{current}$ and $STP_{log}$, two types of resource variations are used between $MD_c$ and $MD_{log}$, as well as $VI_c$ and $VI_{log}$, respectively. We denote by *inter-type variation*, the difference between the resources capacities when two distinct mobile devices or Amazon EC2 instances are compared. In Table 5.1, we compare the capacities of CPU and network (i.e., 3G and WiFi for mobile devices and I/O for virtual instances) resources between the current (Motorola Droid, Amazon EC2 large instance) setup and the logged (HTC Desire, Amazon EC2 small instance) setup.

In practice, the *inter-type variation* does not accurately reflect the difference

**Table 5.1:** CPU and network resource capacities comparison between (Motorola Droid, Amazon EC2 large) and (HTC Desire, Amazon EC2 small) setups.

| | LOGGED | | CURRENT | |
|---|---|---|---|---|
| Resources | HTC Desire | Small instance | Motorola Droid | Large instance |
| CPU | 1 GHz | 1 (x 1 CU) core | 600 MHz | 2 (x 2 CU) cores |
| 3G HSDPA<br>3G HSUPA | 7.2 Mbps<br>2 Mbps | –<br>– | 10.2 Mbps<br>5.76 Mbps | –<br>– |
| WiFi | 802.11 b/g | – | 802.11 b/g | – |
| I/O | – | moderate | – | high |

between two distinct mobile devices or virtual instances. For mobile devices, this is due to the fact that mobile platforms usually run several applications and processes simultaneously, each consuming a fraction of the available CPU and network bandwidth. This means that a mobile device does not actually have 100% availability from the CPU and network capacities to allocate for the incoming application. Therefore, we define the *intra-type variation* that accounts for the actual resource availability on the underlying infrastructures. However, it only allows to compare mobile devices and it does not apply to virtual instances, because in real world scenarios a third party has no access to information about the resource usages on the underlying physical machines. To accurately quantify the differences between the mobile devices and virtual instances belonging to the compared setups, we need both *inter-* and *intra-* type variations.

*Variation coefficients for mobile resources.* First, we combine the variations to quantify how much faster or slower would the mobile device in the candidate setup be relative to the logged device, relative to computational and data transfer operations, as follows:

$$V_{MD}(r)_{cf} = \frac{r_{ca}(MD_c) * (1 - r_{used}(MD_c))}{r_{ca}(MD_{log}) * (1 - r_{used}(MD_{log}))} \tag{5.3}$$

where $r_{ca}$ is the capacity of the resource $r$ (i.e. CPU) and $r_{used}$ represents how much of $r_{ca}$ is already used by other applications or processes, $0 \leq r_{used} \leq 1$. If $V_{MD}(r)_{cf} < 1$, then for those application operations that require resource $r$, the current mobile device would require a longer processing time. The opposite applies for $V_{MD}(r)_{cf} > 1$. For instance, let us consider the HTC Desire in Table 5.1 has 70% CPU usage, while the Motorola Droid has 20% CPU usage. Applying Equation 5.3 we obtain that, even though HTC Desire has a larger CPU capacity, due to its higher usage compared to the Motorola Droid, it would be slower in performing computation operations. Clearly, for such scenarios, varia-

tions are especially emphasized for computationally intensive applications, with the assumption that all available CPU is allocated to the corresponding process.

Since for mobile devices, the network bandwidth provided for upload considerably differs from the one for download, we compute the variation coefficient for each individual operation. In the baseline model, we do not differentiate between request and return transfers along the same communication channel, which means the network variation coefficient is computed as a smoothed average over the upload and download coefficients, as follows:

$$V_{MD}(net)_{cf} = \frac{V_{MD}(net)^{U}_{cf} + V_{MD}(net)^{D}_{cf}}{2} \tag{5.4}$$

$$V_{MD}(net)^{U}_{cf} = \frac{net^{U}(MD_c)}{net^{U}(MD_{log})} \qquad V_{MD}(net)^{D}_{cf} = \frac{net^{D}(MD_c)}{net^{D}(MD_{log})}$$

where the network upload and download speeds for the logged mobile device are computed as medians over collected measurements and the corresponding speeds for the current device are provided from measurements.

*Variation coefficients for virtual instance resources.* In computing the variation coefficients for virtual instances, the following remarks need to be considered: (a) the applications we consider are not parallelized, and therefore cannot take advantage of the multiple cores offered by certain instance types; (b) the maximum upload and download speeds are bounded by the mobile device network interface, which means that the network capacities of the virtual instances can be ignored for the variation coefficients.

Therefore, our proposed approach to compute the variation coefficients for virtual instances considers only the CPU resource and is coarser grained than the model for mobile devices, since the information about the actual resource usages on the underlying infrastructure is not available. We obtain the following equation to compute the coefficient for CPU:

$$V_{VI}(CPU)_{cf} = \frac{n_{cu}(VI_c)}{n_{cu}(VI_{log})} \tag{5.5}$$

where $n_{cu}$ is the number of compute units offered by the virtual instance. If $V_{VI}(CPU)_{cf} > 1$, the remote application bundles would execute faster on the new virtual instance. The opposite applies for $V_{VI}(CPU)_{cf} > 1$. In the example from Table 5.1, switching from an EC2 small instance to a large one would trigger the bundle processing to finish in 50% of the initial time. In practice, this approximation is not extremely accurate. An alternative to improving this estimation would be to consider the real resource usages of the physical machines used for the deployment of virtual instances in the cloud.

*Extending the baseline algorithm.* To increase the accuracy of the baseline derivation for application response times, we need to introduce the computed

variation coefficients for CPU and network resources. The only required change in Algorithm 1 is to replace line 3 with

$$D_i \leftarrow V_i(\frac{1}{V_x(CPU)_{cf}} * S_i + V_{MD}(net)_{cf} * W_i) \tag{5.6}$$

where the service time at the application module $M_i$ is multiplied with the computed variation coefficient for CPU, either on the mobile device (Equation 5.3) or on the virtual instance (Equation 5.5) depending on the bundle location. In addition, the waiting time is multiplied with the coefficient for the network interface on the device, as computed in Equation 5.4.

*Multiple logged setups.* In quantifying the *inter-* and *intra-* variations, we made the simplifying assumption that there is only one logged setup to compare the current execution setup against. In practice, this is hardly the case and therefore it is important to identify from all logged setups the one that is closest to the candidate setup in terms of its resource capacities and usages.

We start by comparing all the logged setups against the candidate setup by looking only at the resource capacities. Let $\mathbb{P}_{log}$ represent the set of all $n$ logged setups, given by their resource capacities. Then $P_c[CPU^c_{ca}, net^{U_c}_{ca}, net^{D_c}_{ca}]$ and $P_{log_k}[CPU^{log_k}_{ca}, net^{U_{log_k}}_{ca}, net^{D_{log_k}}_{ca}] \in \mathbb{P}_{log}$ represent two points in a 3D space corresponding to the current and $k$-th logged mobile devices ($1 \leq k \leq n$), respectively. Each point consists of the capacities of the CPU, network upload and download resources. We quantify the difference between the current and the selected logged device by computing the mathematical distance between the corresponding points as follows:

$$D_{MD}(P_c, P_{log_k}) = \tag{5.7}$$
$$\sqrt[2]{(CPU^c_{ca} - CPU^{log_k}_{ca})^2 + (net^{U_c}_{ca} - net^{U_{log_k}}_{ca})^2 + (net^{D_c}_{ca} - net^{D_{log_k}}_{ca})^2}$$

Similarly, for virtual instances we define the points $P_c[CPU^c_{ca}]$ and $P_{log_k}[CPU^{log_k}_{ca}]$, which only consider the capacities for the CPU resource (i.e., the network capacities on the virtual instances are exceeding those of the mobile devices, which actually bound communication transfers) as follows:

$$D_{VI}(P_c, P_{log_k}) = \sqrt[2]{(CPU^c_{ca} - CPU^{log_k}_{ca})^2} \tag{5.8}$$

For each logged setup, we compute both distances $D_{MD}$ and $D_{VI}$ relative to the candidate setup and select that logged setup for which the distances average is minimum, in order to further obtain the variation coefficients.

Next, we additionally consider the actual resource usages apart from their capacities, to perform a more accurate comparison. Therefore, for each logged setup we compute the variation coefficients relative to the candidate setup as

defined in Equations 5.3 – 5.5, and choose that setup for which the average of the coefficients and the previously computed distances is minimum. In this sense, we ensure to choose a logged setup for which the service and waiting times, used in the estimation algorithm, would not have to be adjusted with high values of the variation coefficients.

### 5.3.3 Modeling Distribution Schemes

Apart from identifying the resource demands of mobile-cloud applications on the underlying resources and computing resource variation coefficients for candidate setups, the distribution scheme represents another important factor in estimating the interaction response time. We extend the baseline derivation from Chapter 5.3.1.2 in Algorithm 4, to additionally account for the application partitioning, in order to better estimate the service and waiting times. The algorithm complexity is O($m * k$), where $m$ represents the total number of modules in the application and $k$ is the number of modules that reside on the mobile device.

Depending on whether module $M_i$ is located on the mobile device or on the virtual instance, the variation coefficient of the CPU resource applied on the service times is different, as seen in lines 4 and 13. How to better estimate the waiting times proves to be more complex. First, if $M_i$ is acquired on the mobile device, then we approximate the waiting time on its corresponding queue to be 0, because any module that depends on it would also be located on the device. This assumption holds according to our definition of *valid configuration* in Chapter 3.3.8.1. Second, if $M_i$ remains on the virtual instance, it is important to identify those depending modules that are remote, since in this scenario the data transfer will not be negligible. In line 10, we compute the waiting time as a product between the network variation coefficient on the mobile device, the average visit ratio $V_{ij}$ and the waiting time $W_{ij}$ from a depending module $M_j$ to $M_i$. Note that $V_{ij}$ represents the average number of visits from $M_j$ to $M_i$ over a time $T$.

Since $W_{ij}$ was not introduced so far, there are two ways to estimate its average value. The first alternative is to gather statistics from previous executions of the application and apply a smoothed average. The second approach is to estimate its average value based on already known variables, such as the number of $M_j$ modules depending on $M_i$, $W_i$, $V_i$ and $V_{ij}$, as follows:

$$W_{ij} \approx \frac{V_{ij} * W_i * \sum_j^{d[M_j,M_i]==1} ne}{V_i},\ where\ ne = 1 \tag{5.9}$$

We observe that the estimation of $W_{ij}$ is accurate when multiple $M_j$ modules send requests to $M_i$ with different visit ratios, whereas in the opposite case it will always provide an average of the actual waiting times. However, its advantage is the easiness to compute $W_{ij}$, without requiring additional code instrumentation to gather the required statistics.

*Separation of upload and download network variation coefficients.* In computing the network variation coefficients on the mobile device, we initially average over the upload and download coefficients. However, in practice the difference between the upload and download speeds experienced for both WiFi and 3G connections is considerable. This means that the average function does not represent an accurate estimation for the waiting times. Therefore, we separate the download and upload coefficients, such that line 10 in Algorithm 2 is replaced as follows:

$$W_i \leftarrow W_i + V_{ij} * (V_{MD}(net)^U_{cf} * W^U_{ij} + V_{MD}(net)^D_{cf} * W^D_{ij}) \tag{5.10}$$

where $W^U_{ij}$ and $W^D_{ij}$ represent the waiting times spent to transfer the request data from the dependent module $M_j$ to $M_i$ and the response data from $M_i$ to $M_j$ as result to the issued request. The current adjustment requires that we differentiate between the amount of data sent through a request and that sent as the response issued to the request. This information is already provided by the application profiler in AlfredO, that uses structural reflection to obtain the data sizes passed as input and output to methods.

**Algorithm 4** Extension to baseline MVA algorithm for an application composed of m modules.

**Input:** $\mathtt{S}_i$, $\mathtt{W}_i$, $1 \leq \mathrm{i} \leq \mathrm{m}$, $\mathtt{P}_{MD}$, $\mathtt{P}_{VI}$, $\mathtt{M}_i$, $\mathtt{d}[M_j, M_i]$ $(i \neq j)$
**Output:** $\mathtt{R}_i$, $\mathtt{R}$

1: **for** $i = 1 \rightarrow m$ **do**
2: $\quad \mathtt{L}_i \leftarrow 0$
3: $\quad$ **if** $\mathtt{M}_i \in \mathtt{P}_{MD}$ **then**
4: $\quad\quad \mathtt{D}_i \leftarrow \mathtt{V}_i(\frac{1}{\mathtt{V}_{MD(CPU)_{cf}}} * \mathtt{S}_i)$
5: $\quad$ **end if**
6: $\quad$ **if** $\mathtt{M}_i \in \mathtt{P}_{VI}$ **then**
7: $\quad\quad \mathtt{W}_i \leftarrow 0$
8: $\quad\quad$ **for** $\mathtt{M}_j \in \mathtt{P}_{MD}$ **do**
9: $\quad\quad\quad$ **if** $\mathtt{d}(M_i, M_j) == 1$ **then**
10: $\quad\quad\quad\quad \mathtt{W}_i \leftarrow \mathtt{W}_i + \mathtt{V}_{ij} * \mathtt{V}_{MD}(net)_{cf} * \mathtt{W}_{ij}$
11: $\quad\quad\quad$ **end if**
12: $\quad\quad$ **end for**
13: $\quad\quad \mathtt{D}_i \leftarrow \mathtt{V}_i * \frac{1}{\mathtt{V}_{VI(CPU)_{cf}}} * \mathtt{S}_i + \mathtt{W}_i$
14: $\quad$ **end if**
15: $\quad \mathtt{R}_i \leftarrow \mathtt{D}_i(1 + \mathtt{L}_i)$
16: **end for**
17: $\mathtt{R} \leftarrow \sum_{i=1}^{m} \mathtt{R}_i$

## 5.4 Modeling Workload

A last important factor that significantly impacts the performance of applications is the *workload*. It is well known that the the interaction mix of applications does not remain constant over time, which makes interactions *nonstationary*. A practical example consists of an application that generates panoramas based on inputed images. In most cases, users submit pictures that are different in their size and number. It is easy to see that in such scenarios the interaction time, which encompasses the data transfer and processing times, fluctuates within a relatively large range, as shown for instance in our experiments with the indoor localization application in Chapter 4.2.6.

One implication of interaction mix stationary is that the full spectrum of workloads for which we must predict performance may not be available during model calibration. Performance models must therefore generalize to workloads that are very different from those used for calibration. Furthermore, a proper validation of the performance prediction model requires nonstationary workloads, since stationary workloads differ qualitatively from real-world workloads. Synthetic workload generators used in benchmarking typically imply first-order Markov models to determine the sequence of transactions submitted by emulated clients. Such examples include the standard TPC-W workload generator (TPC05). This approach cannot yield the kind of nonstationary workloads observed in real applications, because first-order Markov processes are stationary (Str05).

Next, we look at the implications of nonstationarity for performance modeling. We consider a scalar model as one that ignores interaction mix in workloads and considers only workload intensity or arrival rate. For example, consider an application whose workload consists of equal numbers of A and B interaction types, where A places heavy demands on the underlying resources and B has light demands. Scalar models may work well if the ration between the two interaction types remains equal. However, such models are likely less accurate if the interaction mix changes, when for instance A interactions double and B remains constant, or vice versa. Therefore, we employ *interaction mix models* that predict application-level performance based on counting interaction types. As discussed in (SKZ07), these models have a number of desired properties: (a) their free parameters have intuitive interpretations, (b) they yield accurate performance predictions under a wide range of scenarios, and (c) the model calibration is fairly straightforward. These properties combined with nonstationarity means that models can be calibrated using only light-weight passive measurements, which adhere to our initial model objectives and are easily collected in real environments.

The following observations on workload hold for mobile-cloud applications: (a) workload consists of *request-reply* interactions; (b) interactions have a limited number of *types* (e.g., browsing, online payment, printing for a ticket machine

application); (c) interaction types influence resource demands; (d) interaction mix is *nonstationary*, meaning it changes over time. The first two observations apply to every distributed application encountered in practice. The third property arises because interaction types often determine the runtime code path through application logic, which in turn strongly influences resource demands. These assumptions may determine one to model the performance by accounting for interaction service times only and ignoring waiting times, as discussed in (Kel05, SS05). However, for the applications we consider waiting times are in most cases non-negligible and the model must explicitly account for these.

Nonstationary processes can be used to model workloads, but do not address the complementary problem of workload forecasting. We make the following assumptions prior to formulating the workload model:

- all application interactions during model calibration are logged
- the CPU utilization of every application module is extracted from the measured execution times (i.e., execution times correspond to the service times $S_i$ at corresponding queues in the model)
- the bandwidth utilization between communicating modules is extracted from measured data transfer times (i.e., based on the queue model, the transfer times are equal to the waiting times $W_i$)

We further describe the workload performance model, with the general high-level form represented as follows:

$$P = F_{\vec{cv}}(\vec{W}) \tag{5.11}$$

where $P$ represents the application performance as response time, $F$ specifies the functional form of the workload model, $\vec{cv}$ is a vector of calibrated parameter values, and $\vec{W}$ is a vector of workload characteristics. Consider that the application is monitored for a time period of length $T$, divided into short non-overlapping intervals. For interval $t$, let $N_{ti}$ denote the number of interactions of type $i$ that began during the interval and $R_{ti}$ denote the sum of their response times. Therefore, the model has the form:

$$\beta_t = \sum_i R_{ti} = \sum_i \alpha_t N_{ti} \tag{5.12}$$

where $\beta_t$ is the sum of all interaction response times during interval $t$ and $\alpha_t$ denotes model parameters that are obtained through model calibration. Let $st_t$ represent the calibrated values, that intuitively represent typical service times for the various interaction types. For the given model parameters $st_t$ and observed interaction mix $N_{ti}$ at time $t$, $\beta_t$ becomes:

$$\beta_t = F_{\vec{cv}}(\vec{N_t}) = \sum_t st_t N_{ti} \tag{5.13}$$

If the $N_{ti}$ interactions represent past workload, $\beta_t$ can be interpreted as the model's prediction of what aggregate response time should have been during interval $t$. If instead the given interaction mix is a forecast of future workload, the estimated performance represents the model's prediction. Further, we need to account for the waiting times as follows:

$$\beta_t = \sum_t (st_t + wt_t) * N_{ti} \tag{5.14}$$

where $st_t$ and $wt_t$ parameters represent the service and waiting times observed during model calibration. To compute the waiting times, we need to consider the amount of data transferred between application modules for an interaction type and any delays that may be caused by high resource utilizations, $d_t$. In the single-user scenario, such delays do not apply, as opposed to multiple-users cases. We discuss how these delays could be computed. Every resource is assigned an M/M/1 queue and $U_{tr}$ denotes the utilization of resource $r$ during time interval $t$. A naive method is to linearly add all resource utilizations over all time intervals, as shown on the left-hand side of Equation 5.15. An alternative is to compute the average waiting times as a combination of resource utilizations and the arrival rate $\lambda_t$, as shown in the right-hand side.

$$d_t = \sum_r U_{tr} \qquad or \qquad d_t = \sum_r \frac{U_{tr}^2}{\lambda_t * (1 - U_{tr})} \tag{5.15}$$

### 5.4.1 Model calibration

To calibrate the model, we consider a data set consisting of aggregated response times $\beta_t$ and interaction mixes $N_t = (N_{t1}, N_{t2}, ...)$. According to (NKNW96), a good model calibration requires roughly ten times more measurement interval $t$ as interaction types. The output of calibration is a set of calibrated parameter values $\alpha_t$ corresponding to the $st_t$ and $wt_t$ parameters.

The goal of model calibration is to compute parameters that maximize its accuracy. To define accuracy we need to compare the actual measured response time and the value obtained from the calibrated performance model. Therefore, if $r_t$ is the actual measured aggregate response time during the interval $t$ and $\beta_t$ is the fitted value obtained from calibration, let $err_t = r_t - \beta_t$ denote the residual error at time $t$. The normalized aggregated error represents the measure of model accuracy and is defined as follows:

$$na_{err} = \frac{\sum_t |err_t|}{\sum_t r_t} \tag{5.16}$$

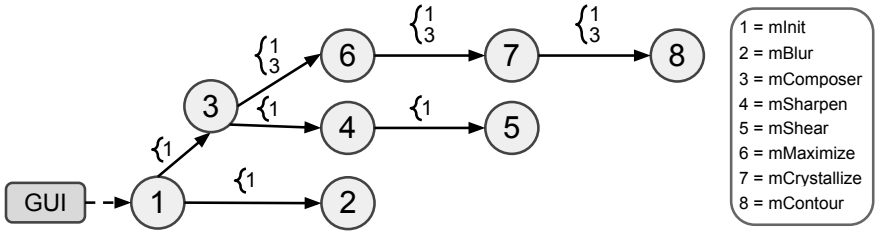



**Figure 5.5:** The modularization scheme for the image processing application.

To achieve optimal accuracy a calibrated model must minimize the sum of residual errors, namely the numerator in Equation 5.16. This is a case of linear programming, where different algorithms, such as (BR73), can be used. The algorithm yields model parameters that optimize retrospective accuracy with respect to the data used for calibration. It is known in common literature as the least absolute regression (LAR) regression method. Ordinary least squares (OLS) regression minimizes the sum of squared residuals, and although it is a cheap method, Kelly et. al (Kel05) shows that such an approach can have worse accuracy than LAR. An additional benefit of LAR is its robustness, meaning it is far less sensitive to distortion by outliers in the data set than OLS. Many variants of robust regression methods are available, some based on OLS and LAR (RT99, Wil05). However, we chose to use the plain-vanilla LAR because it is conceptually simple and it involves no tunable parameters. A final benefit of using linear programming to achieve model calibration is that it becomes easy to add additional constraints on the parameter values.

## 5.5 Experimental Evaluation

To validate our performance estimation model and evaluate its accuracy, we consider an image processing application, modularized as shown in Figure 5.5 and adapted from the interior decoration application described in Chapter 4.1. We evaluate the accuracy of the model with two input images of 1MB and 2MB, such that the data transfers vary between 1.5MB and 6MB. Additionally, certain application modules are visited multiple times. More specifically, the `mMaximize` module receives `k` requests, where `k` is determined by the number of times the module needs to be executed on the smaller image until its size doubles. The value of `k` depends both on the maximization factor (i.e., set to 0.4) and on the effects the `mCrystallize` and `mContour` modules have on the image size. We observe that in practice, for modules `mMaximize`, `mCrystallize` and `mContour`, `k = 3` for most input images.

Our evaluation methodology follows several steps. First, the application is executed in AlfredO for all its 27 valid distributions (i.e., starting from keeping only

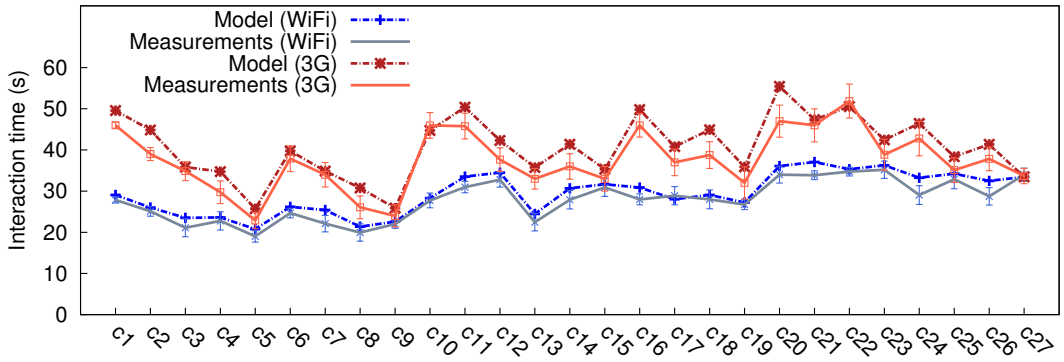



**Figure 5.6:** Comparison between estimated interaction times and collected measurements for all 27 valid distribution schemes for the image processing application.

the user interface on the mobile device and gradually migrating modules, until the application is fully acquired on the client) and measurements are collected over 10 such runs, when an HTC Desire in idle mode and a small EC2 instance are used. HTC Desire runs Android 2.1, has a Qualcomm QSD 8250 1 GHz processor and 576 MB of RAM. The smartphone communicates with the cloud using WiFi (IEEE 802.11 b/g) or 3G. The small EC2 instance has 1.7GB RAM, 1 core and 169GB storage. For validation, the model uses the gathered statistics to estimate the response times, that are further compared to the measured ones. The model accuracy is evaluated against collected measurements in several scenarios where impacting factors for performance are varied: (a) the small instance is replaced with an large one; (b) the CPU load on the HTC Desire is increased to 50–60%; (c) the available bandwidth on the HTC Desire is decreased, by simultaneously running a media application; (d) the HTC Desire device is replaced with a Motorola Droid, and its CPU load is increased again to 50–60%; (e) user interaction mixes are nonstationary. In all (a–d) experiments, the model uses the initial statistics from running the application with the HTC Desire and a small instance, while the workload experiment requires model calibration. Finally, we compare the distributions chosen by the model to those obtained with AlfredO in two cases: (a) extensive profiled data is available for all execution scenarios, and (b) initial statistics are used only.

### 5.5.1 Model Validation

To validate the model, statistics are gathered from running the application in all possible distributions on an HTC Desire device in idle mode (i.e., no additional foreground or background applications are running simultaneously) and a small EC2 virtual instance. The model estimations are obtained by running the derivation algorithm on the averages of the processing times at application modules, the

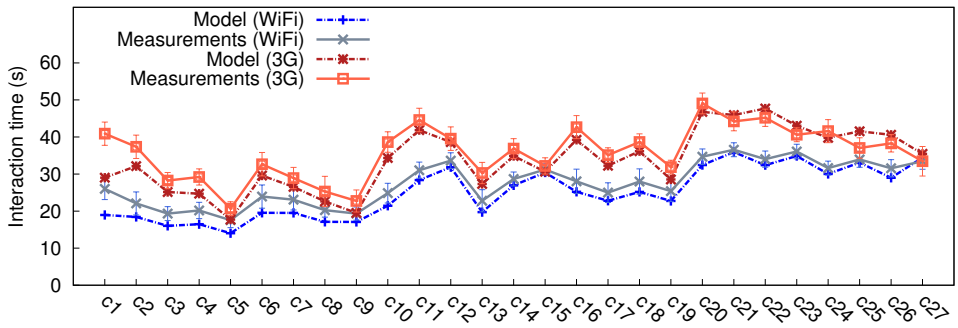



**Figure 5.7:** Comparison between estimated interaction times and collected measurements for all 27 valid distribution schemes for the image processing application on a large EC2 instance.

data transfer times between communicating modules, the download and upload speeds, as well as the visit ratios.

These estimations are further compared to the real measurements and results are shown in Figure 5.6. As observed, the estimations and the measurements follow similar trends and their values are close. When comparing the real interaction times with the estimated ones, the WiFi errors produced by the model are below 5% and for 3G below 7%. These results, where the WiFi errors are lower by 2–3% than those for 3G, correspond to our expectations that in practice WiFi connections are more stable. Additionally, we also notice that our model underestimates the observed interaction time, which is a direct effect of the approach used to compute data transfer times.

### 5.5.2 Adaptation to Virtual Instances

Further, we consider the scenario where the HTC Desire device in idle mode is used with a large instance, which has 2 cores and each core contains 2 computing units. Our image processing application is not parallelized, therefore only one core is used for remote execution. However, compared to the small instance, the large one still offers double the number of computing units. The model adapts the variation coefficient to reflect the change in the CPU capacity of the virtual instance. As a result, the processing time of the application modules executing remote is estimated to decrease by 50% relative to the gathered statistics with the small instance. In practice, this estimation is too coarse-grained and one would not experience such significant differences in performance between small and large instances. A primary factor is the lack of knowledge of the resource usages on the underlying physical machines used for deployment.

Figure 5.7 shows the estimated interaction times and the real measurements for all partitionings. In the WiFi scenario, the model errors in interaction times

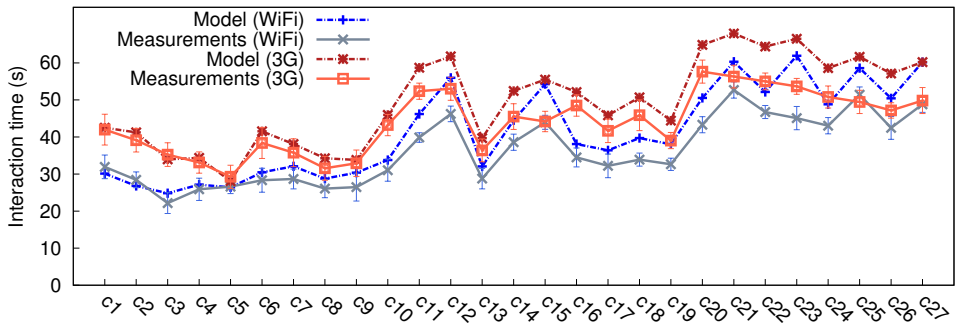



**Figure 5.8:** Comparison between estimated interaction times and collected measurements when the CPU load on HTC Desire is raised to 50–60%.

are below 25% and for 3G below 28%, with averages of 8% and 10%, respectively. Although these errors are not negligible, the model finds the optimal distribution configuration, namely `c5`, as reported by real measurements. The optimal configuration contains the `mInit`, `mBlur` and `mComposer` modules. Looking more closely to the results, the underestimation of the model is more significant for the first configurations. This is an expected effect, because most of the application modules are executed in the cloud and, thus, they are more significantly affected by the underestimation of the variation coefficient.

### 5.5.3 Adaptation to Device CPU Load

In the next scenario, we evaluate the model accuracy when the HTC Desire device is already running additional background processes that increase its CPU load to 50–60%, prior to start interacting with the image processing application. With a higher CPU usage, we expect the application modules running locally to require a longer processing time by a factor estimated by the model to be around 1.8. In practice, computing an accurate coefficient of the CPU variation becomes difficult, especially because of complex context switching given by the operating system scheduling strategy and the dynamic process priority ranking.

Figure 5.8 compares the estimated interaction times and the real measurements. The model errors in the WiFi scenario are below 24% and for 3G below 26%, with averages of 12% and 14%, respectively. However, the model correctly estimates the configurations `c3` (i.e., composed of the `mInit` and `mBlur` modules) and `c5` to be the optimal distributions for WiFi and 3G. We expect that increasing the CPU load on the device determines the migration of certain local modules to the cloud, to speedup up the execution. This is the case for WiFi, where the higher upload and download speeds allow for larger data transfers. Instead, for 3G the performance improvement from cloud execution is less significant than the time spent in remote communications, and therefore more application modules

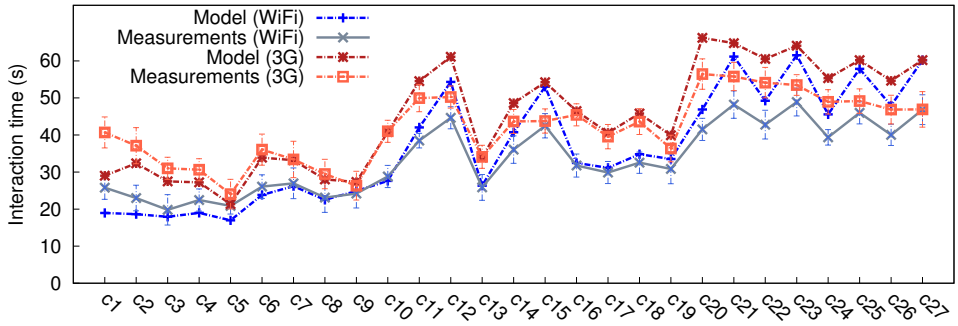



**Figure 5.9:** Comparison between estimated interaction times and collected measurements when the CPU load on HTC Desire is raised to 50–60% and a large instance is used.

run on the device. Further we make the following notes: (a) the model overestimates the interaction times, as opposed to the case where the virtual instance has more computational capability, and (b) the overestimation is more significant for the last configurations, where most or all application modules execute on the mobile device.

### 5.5.4 Adaptation to Virtual Instances and Mobile Devices

Next, we evaluate how our performance model adapts when variations for both the mobile device and the virtual instance are introduced. We consider the following three scenarios: (a) the small instance is replaced with a large one and the CPU load on the HTC Desire device is increased to 50–60%, (b) the HTC Desire device is replaced with a Motorola Droid and its CPU load is increased to 50–60%, and (c) the available bandwidth on the HTC Desire is reduced and a large instance is used. The Motorola Droid smartphone runs Android 2.1, has a 600 MHz Cortex-A8 processor and 256MB of RAM.

In the first scenario, we formulate two assumptions. On the one hand, we expect more significant differences between the real and estimated response times for those configurations where most or all modules are either executed on the device or remotely on the virtual instance. More precisely, the model should underestimate for the first configurations (i.e., `c1` to `c5`) and overestimate for the last ones (i.e., `c21` to `c27`). On the other hand, we expect smaller such differences for configurations that balance the distribution of modules between the client and the cloud (i.e., `c7` to `c18`). These assumptions are sustained by the results shown in Figure 5.9. Given the rather coarse-grained computation of the variation coefficients, the differences between the real and estimated times are indeed larger at the extreme distributions. However, for the balanced distributions this difference is at times larger than expected, due to the network variations. In

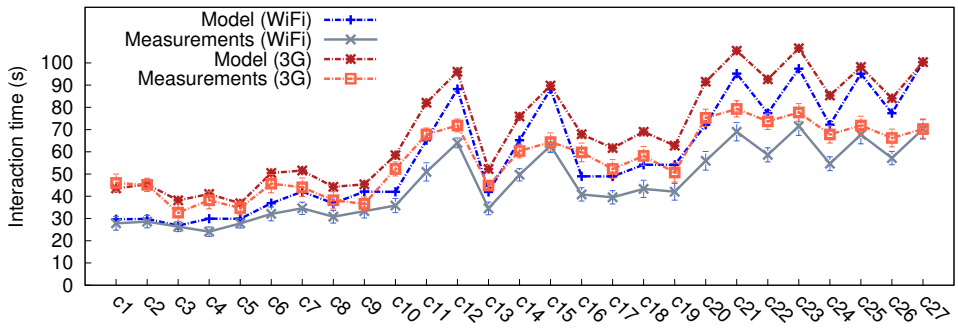



**Figure 5.10:** Comparison between estimated interaction times and collected measurements when the HTC Desire is replaced with a Motorola Droid and its CPU load is increased at 50–60%.

the WiFi scenario, the model chooses the configuration `c5` when the optimal one is `c3`, while for 3G `c5` is correctly chosen. The penalty in choosing a sub-optimal configuration for WiFi is under 3 seconds and is due to the small differences between response times of certain distributions and the standard deviation. In the 3G case the time difference between `c5` and the other distributions is more significant, making it easier for the model to decide on the optimal configuration.

In the second scenario, the HTC Desire device is replaced with a Motorola Droid. Mobile device kernels dynamically adjust the minimum and maximum CPU frequencies to save battery or reduce execution times of applications, for instance. To obtain more accurate model estimations based on the variation coefficient for the device CPU, we manually set the frequency minimum and maximum values for the duration of the experiment. On Android, this is achieved by using the SetCPU (CPU09) application with root privileges. On both devices the minimum and maximum frequencies are set to their specified capabilities. While the process of manually setting the frequency runs smoothly on the Motorola Droid, the HTC Desire uses the `perflock` driver to prevent an application from changing frequencies. We solved this problem by disabling the driver. The comparison between the estimations and the real measurements is shown in Figure 5.10. The model overestimates response time by even 25 seconds for WiFi when most or all modules are executed locally. The maximum error is below 37%, while the average error is around 20%. In the case of 3G connections, the model overestimates by at most 30 seconds and the maximum error is below 41%, with an average error of 24%. Although these errors are not negligible, the model manages to choose a nearly optimal distribution, namely `c3` for WiFi (i.e., `c4` = (`mInit`, `mComposer`) is the optimal configuration) and `c5` for 3G (i.e., `c3` is optimal). The penalty in performance produced by the model is under 3 seconds for WiFi and under 5 seconds for 3G.

Finally, we consider the scenario where the available bandwidth on the de-

vice is reduced by running simultaneously a media application that uploads and downloads images. For WiFi, the bandwidth is reduced from 4.5Mbps to around 2.5Mbps, while for 3G from 2Mbps to 900Kbps. In both cases, the desire is to minimize the data transfers and the estimation model chooses the configuration `c9` = (`mInit`, `mBlur`, `mComposer`, `Sharpen`, `mShear`) as the optimal one. Indeed, for WiFi the measurements show that `c9` provides the shortest interaction. For 3G the optimal distribution is still `c5`, even with the reduced bandwidth, but the difference in times between `c5` and `c9` is under 6 seconds. These experiments show that the model is able to choose the optimal or nearly optimal distribution of the application. Additionally, the user cost in performance is only in the order of seconds and can be further improved after the first interaction, by using the new statistics to adjust the average module execution and data transfer times.

### 5.5.5 Comparison to AlfredO

Finally, we consider the previous scenarios and compare the model estimations with AlfredO running in its default mode (i.e., profiled statistics are available only for the default execution environment, HTC Desire in idle mode and small virtual instance) and its ideal mode (i.e., profiled statistics are available for all scenarios), respectively. When compared to the real measurements, AlfredO in ideal mode always decides on the optimal distribution, thus it represents the ground truth. Results are shown in Table 5.2.

**Table 5.2:** Comparison between AlfredO ideal, the model and AlfredO default.

| Scenario | AlfredO ideal | | Model | | AlfredO default | | | |
|---|---|---|---|---|---|---|---|---|
| | WiFi | 3G | WiFi | 3G | WiFi | Imp. | 3G | Imp. |
| EC2 large | c5 | c5 | c5 | c5 | c8 | 17% | c5 | 0% |
| EC2 large + CPU load | c3 | c5 | c5 | c5 | c4 | 12% | c4 | 27% |
| Droid + CPU load | c4 | c3 | c3 | c5 | c2 | 14% | c2 | 32% |
| EC2 large + bandwidth | c9 | c5 | c9 | c9 | c7 | 15% | c3 | 29% |

Column `Imp.` reports by how much the model improves the application interaction time compared to AlfredO default. To obtain the improvements, we first compute the difference between the response times with the chosen configurations by AlfredO default and the model. Further the resulting difference is divided by the optimal interaction time, given by measurements. The improvements are higher for 3G, except for the scenario where only the small instance is replaced by a large one, and for both WiFi and 3G they are in the order of seconds or tens of seconds, already for single user interactions. This means that for multiple interactions, which is the norm for many applications, the improvements become

even more significant. Thus, such a simple model has the potential of providing users with better experiences, while requiring only passive measurements.

### 5.5.6 Adaptation to Workloads

To understand how the performance model adapts to workload variations, we calibrate the model and evaluate its retrospective accuracy on the first half of the validation data set. Further, we apply the calibrated model to the interaction mix in each time interval $t$ of the second half to obtain the values for $\beta_t$, as defined in Equation 5.14. Finally, these estimated values are compared to those observed through real measurements. The environment used for calibration consists of an HTC Desire device in idle mode and a small EC2 instance, while for the second half of the data set a large instance is used instead. The application used for calibration and evaluation is the same image processing application as before, while the data set used consists of a sequence of different interaction types and is summarized in Table 5.3. To obtain different interaction types, we vary the sizes of the inputed images from 100KB to 1.5MB and from 200KB to 3MB, respectively, as well as the number of such pairs from 1 to 3. The total number of interactions for both WiFi and 3G scenarios is 200 and interaction types are uniformly distributed. As seen in the summary table, the standard deviation of response times over all interaction types is significant and shows the importance of considering the nonstationary character of interactions.

**Table 5.3:** Summary of data set used for calibration and test phases.

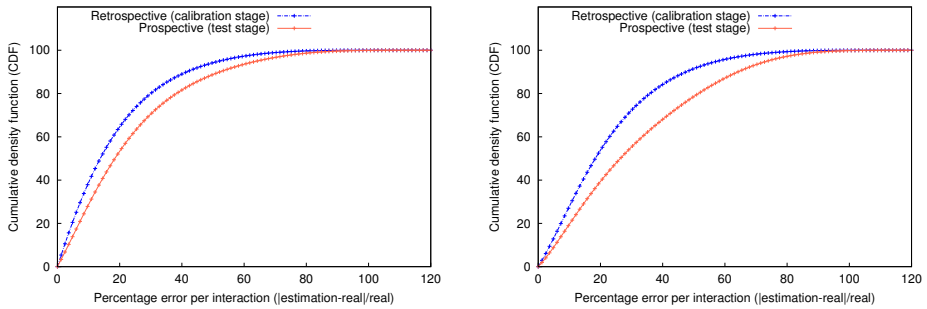
| | Duration | Interactions | | Response time | | |
|---|---|---|---|---|---|---|
| | | Number | Types | Mean (sec) | Median (sec) | St. dev. |
| WiFi | 93 minutes | 200 | 34 | 26.26 | 23.2 | 16.41 |
| 3G | 121 minutes | 200 | 34 | 34.32 | 30.11 | 19.98 |

The model is calibrated on the first half of the data set by executing each interaction with the optimal configuration from the validation experiment, namely `c5`. Further, we evaluate the accuracy of the calibrated model based on Equation 5.16 and using the interaction mixes in the second half of the data set. The nonstationarity character of the interaction mix implies that workloads in the second half can be different than those used in the calibration phase. Table 5.4 presents the retrospective accuracy of the model when calibrated with both OLS and LAR on the data set.

The average error is relatively low while considering both WiFi and 3G, under 19% for LAR and 24% for OLS, even when the data set contains interaction types that are more computational intensive. Additionally, we observe that the error produced is lower when LAR regression is used. Some outliers exist, especially

**Table 5.4:** Retrospective and prospective accuracy on the first half and second half of the data set, respectively.

| Calibration method | Retrospective accuracy | | Prospective accuracy | |
|---|---|---|---|---|
| | WiFi | 3G | WiFi | 3G |
| OLS | 19.11% | 23.13% | 32.19% | 37.3% |
| LAR | 15.78% | 18.29% | 27.14% | 30.62% |




(a) WiFi (b) 3G

**Figure 5.11:** CDF percentage error per time interval $t$ for both WiFi and 3G.

when images with larger sizes are used, but LAR provides robust estimation and is less sensitive than OLS, and therefore yields more accurate models according to the accuracy measure used.

Table 5.4 also summarizes the prospective accuracy of the model on the second half of the data set. Prior to discussing the results, we make several observations. Given the variation in interaction types, configuration `c5` is not necessarily the optimal distribution partitioning in all scenarios. Therefore, to show the model accuracy and its adaptation to workload variations, an intermediary step is introduced after calibration. The gathered statistics for module execution times and waiting times, as well as the calibration parameters are used with the model defined in Chapter 5.3.3 to estimate the optimal or nearly optimal distribution scheme for particular interaction types. The assumption is there is no forecasting involved, which means these interaction types are known. Further, the resulting estimations are cached on the mobile device, thus allowing it to autonomously decide how to distribute application modules for each incoming workload in the second half of the data set. Results for the prospective accuracy show higher errors on the second half of the data set, under 31% for LAR and 38% for OLS. This is due to several factors. First, given the nonstationarity property of interactions, some interaction types do not occur during model calibration. Second, using the performance model to estimate good distribution schemes introduces an additional error. Third, due to the higher variation in 3G connections and

its direct impact on the workloads, we observe that the errors are higher for 3G, which is consistent with the model calibration as well. Figure 5.11 illustrates the distributions of $\frac{err_t}{r_t}$ for both calibration and test phases. Results show that the residuals remain quite small in the WiFi scenario when the model is used on the second half of the data set in comparison with the calibration step. For 3G, the residuals become more significant, which is a cumulative effect of using a different EC2 instance in the test phase, the variability of 3G connections and estimating good distribution schemes with the performance model.

Finally, real measurements on the second half of the data set show that optimal configurations change according to the image sizes. For WiFi, `c3` and `c4` are best when the size of the large image is less than 1.2MB, while `c5` is optimal for all interactions in which the size of the large image is under 2.5MB. For larger sizes, `c9`, `c13` and `c19` are optimal. For 3G, we notice that `c3` is best only for image sizes under 500KB, while `c5` is optimal when the size of the large image is less than 2.2MB. For larger sizes, `c9` and `c19` are optimal. In comparison, the model estimations are quite accurate especially for smaller sized inputs. For certain larger images it introduces `c15` as being the optimal distribution for both WiFi and 3G, while for others it finds nearly optimal configurations. The penalties in response times due to these estimations are under 12 seconds for WiFi and 20 seconds for 3G, in the conditions when the corresponding optimal interactions require at least 40 seconds, respectively, 55 seconds.

## 5.6 Summary and Discussion

The proposed model uses queue networks to model the structure and behavior of modular mobile-cloud applications. We identify relevant factors that impact interaction times, such as application distribution, resource demands, execution environment or workloads, and combine them into a unified solution. Our evaluation shows that the model requires only passive measurements to choose the optimal or nearly optimal application distribution, and thus provides users with good experiences at a lower cost than AlfredO in the ideal scenario. The queue model has several desirable features, such as simplicity and generality. The generality property comes from the model's ability to handle applications with an arbitrary number of modules. The MVA algorithm works without making any assumptions about the service time distributions of the clients. This feature is desirable because it is representative of scheduling policies in mobile and commodity operating systems, and it implies that the model is general enough to handle workloads with arbitrary service time requirements.

However, we note that our model does no explicitly capture certain scenarios, such as modeling multiple resources at module level. The reason is due to fact that each application module is represented as a single queue, while in practice

the underlying mobile device or physical machine exhibits various resources such as CPU, disk or memory. Currently, the model does not capture the utilization of the various resources by a request at a specific module. An enhancement to the model where these resources are modeled as a network of queues is the subject of future work. Additionally, the model does not currently support multiple clients. As discussed in Chapter 5.3.1, the only viable solution is to improve performance in a global fashion, thus for all clients entering the system within a time frame. However, such an approach generates a large combinations space and makes the problem intractable. Another scenario our model is not considering consists of resources being held at multiple modules simultaneously. This is because it essentially captures the flow of an interaction through the application modules as a sequence of time periods, during which the request utilizes resources at exactly one module. Although this assumption is valid for many mobile-cloud applications, it does not apply to parallel applications, where in order to minimize response time, sub-requests activate multiple modules. An example is document translation applications, where paragraphs are translated simultaneously at replicated modules. A further limitation of the model is not considering the underlying resource usages on the physical machines used to deploy virtual instances. Clearly, knowing how many additional instances are simultaneously using the same set of resources and how many of these resources are already demanded elsewhere can improve the accuracy of the model.

# Chapter 6

# Efficient Application Placement in Cloud Data Centers

In the context of integrating cloud-based applications with mobile devices, we have so far considered application complexity from the mobile perspective only. Our goal was to provide the optimal partitioning of such applications between a mobile device and a cloud instance such that users experience lower interaction times and extended battery life. However, the increasing application complexity needs to be addressed from the cloud perspective as well. Applications deployed in data centers no longer require only one virtual instance to execute efficiently. Instead, multiple instances per application with compute and network demands, as well as performance and robustness related constraints, are slowly becoming the norm. Therefore, we move for the simpler problem of distributing applications between two-party systems to the more complex one that considers multiple-party systems. In our context, this relates to efficiently distributing the application part residing in the cloud between multiple physical machines, while satisfying its complex demands and constraints. In this chapter we propose such a placement technique and validate it with a rich set of workloads.

## 6.1 Motivation

In enterprise data centers, infrastructure architects tailor hardware and software configuration to optimize for their workloads. To run a production application, the administrator provisions physical machines, storage, networks, middleware, and application code such that the application is resilient to hardware failures and performance bottlenecks. The cloud changes the infrastructure provisioning model. A typical IaaS offers virtualized building blocks, such as virtual machines, storage volumes, and networks, which users of the cloud connect together to create virtualized infrastructures for their workloads. Very little control is given

to a user with respect to the layout of these virtualized building blocks on the physical infrastructure. As a result, it is impossible for the user to build a virtualized infrastructure that guarantees, for instance, high communication bandwidth between virtual machines, proximity to storage or spreading of multiple virtual machines across different racks for availability reasons. As a matter of fact, the only support for workload optimization available in today's cloud is via pre-built virtual infrastructures which are tuned to specific workloads. For instance, Amazon EC2 offers high performance computing (HPC) instances (HPC12).

We believe that this cookie-cutter approach hinders further adoption and development of the technology. Instead, cloud users should be able to design and deploy virtual infrastructures that optimize for their workload. We refer to these virtual infrastructures as virtual network infrastructures (VNI). More specifically, a VNI consists of a collection of heterogeneous virtual machines with compute and network demands, as well as constraints governing their performance as a whole in order to satisfy application requirements. We address one crucial problem in enabling this vision, that of developing placement techniques that allow the cloud to efficiently and effectively allocate resources that satisfy VNI constraints, as well as cloud level goals.

A VNI can be represented as an attributed graph. As such, the VNI placement problem is equivalent to the problem of graph monomorphism and therefore is NP-hard (Ben02). The complexity of the problem arises from its combinatorial nature thus, efficiency is a major challenge. Others in the community have tackled less constrained versions of this problem (LLM11, BASS11, MPZ10). The proposed approaches however, address the placement problem in a piecemeal fashion: they either focus on the aspects pertaining to network performance leaving aspects of the allocation of compute resources as a secondary objective or vice-versa, or suffer from high complexity. On the other hand, our proposed approach (GCTS12) tackles the problem in a comprehensive manner by factoring in network, compute and availability performance aspects into one single solution. Considered applications consist of a multiplicity of virtual machines connected via a pair-wise connectivity model. Virtual machines are associated with computing constraints, such as computational power, disk storage or memory, while pair-wise communication links are associated with bandwidth requirements. High-availability and high-communication constraints are translated to anti-collocation and collocation constraints, respectively. The former allows the cloud to contain the impact of failure-locality, while the latter allows it to provide good network performance to users. Such constraints are especially important for mobile users, that are more susceptible to performance degradations in the cloud.

Our placement technique, implemented as a standalone framework, makes the problem tractable and is generic enough to support increasing complexity. The main focus is the introduction of a data center-level resource abstraction, called a cold spot from here on. Such cold spots consist of collections of physical

computing nodes that exhibit high availability of compute resources and network connectivity. The objective of using such an abstraction is to facilitate the identification of subsets of resources where incoming VNIs should be best deployed. Therefore, cold spots are meant to reduce and guide the search space for the optimization problem. Our placement technique is divided in four tractable steps: (a) discovering cold spots, (b) clustering virtual machines to reduce overall communication traffic and placement complexity, (c) identifying candidate cold spots whose features are similar to those of the VNI in order to increase the chances of deployment, and (d) performing the actual placement by using efficient graph-based search algorithms that optimize for load balancing. We validate the proposed approach with different workloads and show its efficiency and effectiveness in medium and large size data centers.

## 6.2 Problem Formulation

Consider a data center which consists of a collection of physical machines (PM) that are inter-connected by a network consisting of a set of links (LK). Every PM can host one or more virtual machines (VM). A VNI comprises a set of networked and constrained VMs and is the deployable unit within the data center. The placement problem consists of mapping the VMs of an incoming VNI to PMs in the data center. Further, we describe the physical infrastructure and VNI characterization in more detail. Table 6.1 summarizes the most common terms used throughout the remainder of this chapter.

Let $\mathbb{PM} = \{PM_i | i = 1, 2, ..., n_{PM}\}$ denote the set of physical machines in the data center. Each PM has a set of resources $\mathbb{R} = \{r_m | m = 1, 2, ..., n_R\}$, such as CPU, memory, and disk storage. The total capacity of resource $r_m$ on $PM_i$ is denoted by $rc_{i,m}$, whereas $ru_{i,m}$ represents its usage on the same PM, $ru_{i,m} \leq rc_{i,m}$. The amount of resource available for $r_m$ on $PM_i$ is defined as $ra_{i,m} = rc_{i,m} - ru_{i,m}$. We assume that a PM is connected to a switch through an edge link, and that switches are interconnected through core links.

The network is modeled as a graph, where PMs and switches are vertices, while links are edges. We denote the set of links as $\mathbb{LK} = \{LK_k | k = 1, 2, ..., n_{LK}\}$. Each link is characterized by a communication bandwidth. The total bandwidth capacity of $LK_k$ is denoted by $bc_k$, whereas $bu_k$ represents its usage, $bu_k \leq bc_k$. The amount of available bandwidth is defined as $ba_k = bc_k - bu_k$. Therefore, a data center $\mathbf{D}$ is characterized by the tuple $\{\mathbb{PM}, \mathbb{LK}\}$.

A VNI $\mathbf{P}$ is characterized by the tuple $(\mathbb{VM}, \Lambda, S)$. The set $\mathbb{VM} = \{VM_j | j = 1, 2, ..., n_{VM}\}$ represents the collection of virtual machines which constitute the VNI. $VM_j$ is characterized by a set of resource demands $rd_{j,m}$, one per each resource type in $\mathbb{R}$. These resource demands are considered when placing a specific VM onto a PM, in order to make sure that there are enough available resources

on the PM to satisfy the VM demands. The communication bandwidth demand between $VM_i$ and $VM_j$, where $1 \leq i, j \leq n_{VM}$ and $i \neq j$, is denoted by $\lambda_{i,j} \geq 0$. We assume that the matrix $\Lambda = [\lambda_{i,j}]$ is symmetrical with zero diagonal. In other words, bandwidth requirements among VMs in a given VNI may be modeled as an undirected graph, where the vertices are VMs and the edges are pairwise communication demands.

$S$ refers to the availability constraints and are explained as follows. We consider a data center which is partitioned into a hierarchy of availability zones, where PMs in the same zone have similar availability characteristics. As an example, a hierarchy of availability zones may be induced by the containment hierarchy of PMs, bladecenters, racks, cluster and data centers. In such case, one may model this hierarchy as a tree, where the leaves are the PMs and the intermediate node represents a zone of availability. Thus, we associate an availability level, $V_l, l = 0, \cdots, L$, for a node at level $l$ in the tree, where $l = 0$ represents the leaves, i.e., PMs, and $l = L$ represents the root of the tree with height $L$, i..e, highest level switch. We assume that $V_0 \leq V_1 \cdots \leq V_L$, since two PMs in distant availability zones have higher chances of having one of them available. Using this tree model, two PMs $PM_i$ and $PM_j$ with the lowest common ancestor at level $l$ have $v_{i,j} = V_l$. Clearly, $v_{i,i} = V_0$. For convenience we define $g_i(l), i = 1, \cdots, n_{PM}$ and $l = 0, \cdots, L$ as the set of PMs such that for $PM_j \in g_i(l)$ we have $v_{i,j} = V_l$. Following these observations, availability constraints can be directly mapped into

| Term | Description | Term | Description |
|---|---|---|---|
| $r_m$ | Resource (e.g., CPU) | $ba_k$ | Bandwidth available in $LK_k$, i.e., $b_{ck} - b_{uk}$ |
| $rc_{im}$ | Total capacity of resource $r_m$ on $PM_i$ | $b_{uk}$ | Bandwidth usage of $LK_k$ |
| $ru_{im}$ | Usage of resource $r_m$ on $PM_i$ | $b_{ck}$ | Total bandwidth capacity of $LK_k$ |
| $ra_{im}$ | Availability of resource $r_m$ on $PM_i$, i.e,. $rc_{i,m} - ru_{i,m}$ | $rd_{jm}$ | Resource demand of $VM_j$ on resource $r_m$ |
| $\lambda_{ij}$ | Network demand between $VM_i$ and $VM_j$ | $n_{PM_{CS}}$ | Cardinality of the set of PMs belonging to cold spot $CS$ |
| $l_{ij}$ | Locality constraint $(\infty, -\infty)$ | $path(i, j)$ | Set of links on path between $PM_i$ and $PM_j$ |
| $n_{VM}$ | Cardinality of the set of VMs belonging to a VNI | $n_{PM}$ | Cardinality of the set of PMs in data center |

**Table 6.1:** Common terminology used in the placement problem.

locality constraints. More specifically, to represent location constraints between VMs, we define the matrix $S = [s^l_{i,j}]$, where $s^l_{i,j}$ represents the type of location constraint between $VM_i$ and $VM_j$, where $1 \leq i, j \leq n_{VM}$ and $i \neq j$ and $l$ refers to the availability zone level required by the constraint. In this paper we assume two distinct types, namely $s^l_{i,j} \in \{+\infty, -\infty\}$, corresponding to collocation and anticollocation, respectively. To illustrate, an anticollocation constraint at the PM-level ($l = 0$) between $VM_i$ and $VM_j$ indicates that VMs must be placed on different PMs and is associated with an infinitely large communication cost between them. Alternatively, a collocation constraint means that the VMs must be placed on the same PMs.

We denote by $\pi(\mathbf{P}, \mathbf{D})$ a particular placement of VNI $\mathbf{P}$ in data center $\mathbf{D}$ and write it as $\pi(\mathbf{P})$ for brevity. $\pi(\mathbf{P})$ is a vector of length $n_{VM}$, where $\pi_j(\mathbf{P})$ is the PM onto which $VM_j$ is placed. The placement process maps every VM in $\mathbb{VM}(\mathbf{P})$ to a particular PM in $\mathbb{PM}$, such that (a) the VM's resource demands are satisfied by the PM, (b) the bandwidth constraints between any two communicating VMs are met by the links of the data center connectivity network, (c) the pairwise location constraints are satisfied.

## 6.3 Placement Goals

The proposed placement technique considers two classes of objectives, namely system behavior objectives (1–2) and performance objectives (3–5) defined as follows

1. *Efficient and scalable placement* – Incoming VNIs should be placed such that (a) the performance of the application is maximized (e.g., fewer hops between communicating VMs reduces network delay), and (b) the placement time scales with the increasing size of the data center.

2. *High VNI acceptance rate* – The placement algorithm must maximize the number of VNIs for which resources are successfully allocated and constraints are satisfied.

3. *Load balancing* – For all placed VNIs, the objective is to balance resource allocation across the data center.

4. *Resource constraints* – We assume that resources are not over-committed. Hence, VNI resource demands must be met by the corresponding resources available on the data center. Formally, $\forall PM_i \in \mathbb{PM}, \ \forall r_m \in \mathbb{R}$,

$$ru_{i,m} \equiv \sum_{\mathbf{p}} \sum_{VM_j \in \mathbb{VM}(\mathbf{p})} rd_{j,m} \ \mathbf{I}_{\pi_j(\mathbf{p}),\mathbf{PM_i}} \leq rc_{i,m}, \tag{6.1}$$

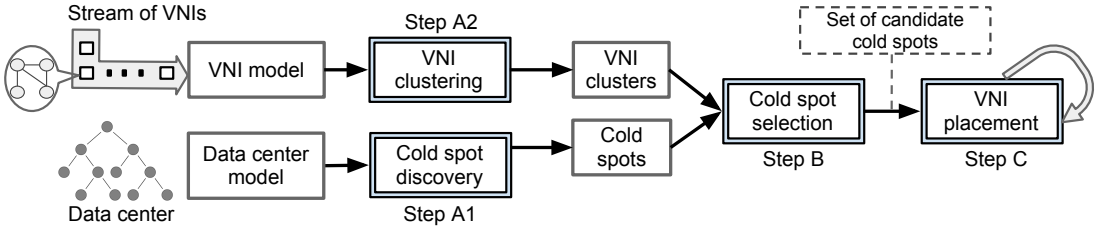



**Figure 6.1:** Placement process of virtual networked applications.

where $\mathbf{p}$ runs over all placed VNI and $\mathbf{I}$ is the indicator function. Furthermore, $\forall LK_k \in \mathbb{LK}$,

$$bu_k \equiv \sum_{\mathbf{p}} \sum_{VM_i, VM_j \in \mathbb{VM}(\mathbf{p})} \lambda_{i,j} \;\; \mathbf{I}_{LK_k \in path(\pi_i(\mathbf{p}), \pi_\mathbf{j}(\mathbf{p}))} \leq bc_k, \tag{6.2}$$

where path$(PM_i, PM_j)$ represents the set of links along the path between $PM_i$ and $PM_j$. For simplicity, we assume that the traffic demand between two PMs is routed through a single path in the network.

5. *Hard location constraints* – The collocation and anticollocation constraints must be satisfied for all placed VNIs. As explained later, softening constraints can be easily relaxed. That is, $\forall \mathbf{p}, \;\; \forall VM_i, VM_j \in \mathbb{VM}(\mathbf{p})$,

$$s^l_{i,j} = +\infty \;\; \Rightarrow \;\; VMj \in g_i(l), \;\; s^l_{i,j} = -\infty \;\; \Rightarrow \;\; VMj \notin g_i(l), \tag{6.3}$$

## 6.4 Placement Algorithms

The proposed approach to meet the constraints and performance objectives presented earlier is to divide the placement problem into four steps as shown in Figure 6.1. Given a data center and a stream of VNI placement requests uniformly distributed, the cold spot discovery (A1) and VNI clustering (A2) steps are performed in an asynchronous manner. Cold spot discovery focuses on identifying cold spots, with the purpose of enabling better resource utilization and placement efficiency. The clustering step, upon receiving a placement request, groups the highly communicating virtual VMs of the VNI, such that traffic is reduced and location constraints are satisfied. Further, the cold spot selection step (B) receives in input the cold spots discovered in step A1, as well as the VNI clusters from step A2 and builds a candidate list of ranked cold spots whose features match the VNI properties. Finally, step C performs the VNI placement within a candidate cold spot. It iterates over the candidate list and attempts to

allocate resources in the current cold spot. If successful, the placement process finishes. Otherwise, the next candidate cold spot in the list is considered. It may happen that no candidate cold spot satisfies the resource and location demands. In such scenarios, our current implementation drops the current VNI and restarts the process for the next placement request.

### 6.4.1 Cold Spot Discovery

A naive approach to VNI placement is to treat the optimizations of computing resources and network as two independent subproblems and solve them sequentially. As shown in (ZA06) such a solution usually results in high utilizations on the links since selected PMs for placement may be located far away from each other. Therefore, we propose an algorithm that integrates both optimizations. One key component of our placement technique is the concept of *cold spot*. A cold spot is a resource construct consisting of a collection of physical computing nodes that exhibit high availability of compute resources and ample network connectivity to other PMs. In principle, any property of interest can be considered when constructing cold spots. This step is concerned with discovering such cold spots in the system and is invoked periodically and asynchronously relative to the placement request. This observation is important since performing any analysis on the data center graph is expected to be computational intensive.

The intuition behind this stage is two-fold. First, such cold spots reduce the search space when placing a VNI within the data center, which results in an overall lower placement time and hence better scalability. Second, they improve resource utilization by reducing resource fragmentation. These benefits are shown with experimental results in Chapter 6.5. The discovery step takes as input the data center model, its state including the currently updated resource allocation, as well as a dynamic threshold parameter and outputs a set of cold spots. This stage is achieved through a coarse-to-fine approach and is further divided into two main steps.

#### 6.4.1.1 Ranking of Physical Compute Nodes

The first step ranks all the PMs in the data center based on their resources availability. We define the availability of a particular $PM_i$ by the measure $RA_{PM_i}$, which is based on the compute resources, as well as the network bandwidth of all outgoing links, and is defined as follows:

$$RA_{PM_i} = \sum_{r_m \in \mathbb{R}} w_m \, ra_{i,m} \;\cdot \sum_{LK_k \in links(PM_i)} ba_k, \tag{6.4}$$

where $w_m$ is a weight which can be adjusted in order to tailor the PMs ranking relative to a specific type of resource and links($PM_i$) represents the set of links

connected to $PM_i$ (via NICs). The above definition captures both the computing resource availabilities of a PM and the network availability of its outgoing links. On the one hand, a high value of $RA_{PM_i}$ means that both the PM itself and some or all of its directly connected links are lightly loaded. On the other hand, highly utilized links or PMs are penalized to lower the $RA_{PM_i}$ value.

We want PMs with ample network connectivity to other PMs in the system to be ranked higher, as they have a higher potential to satisfy the needs of VNIs. This is especially important, for instance, in common scenarios where two communicating VMs are placed on different PMs. In this case, their corresponding communication demand goes through the links connecting the two PMs, which means poorly connected pairs of PMs should be avoided. To this end, we propose a heuristic that has a long-sighted view of the PMs network connectivity. That is, to compute the rank of a given PM the heuristic first identifies the PM's neighborhood as the set of PMs that are K hops away, and then computes its network connectivity to these PMs as a function of network bandwidth. K is a parameter which could be set in relation to the data center diameter or VNI size. This step generates the list of all PMs in decreasing order of their availabilities, computed with the heuristic $NRA_{PM_i}$ defined as follows:

$$NRA_{PM_i} = \frac{\sum_{PM_j \in neighbors(PM_i,K)} \frac{RA_{PM_i}+RA_{PM_j}}{2} min_{LK_k \in path(PM_i,PM_j)}(ba_k)}{\mid neighbors(PM_i, K) \mid} \quad (6.5)$$

where neighbors($PM_i$, K) is the set of PMs that are at most K hops away from $PM_i$. Notice that the heuristic accounts for the minimum available bandwidth on the path between $PM_i$ and a neighbor $PM_j$ to characterize the network component, as well as for both $PM_i$'s and $PM_j$'s compute resource availabilities. This definition comes in contrast with a proposed heuristic in (ZA06), which considers network congestion at the PM level only. Thus, a PM with high CPU, disk storage and memory available, as well as considerable unused bandwidth on its NIC will be ranked high although it may be sitting behind a highly congested switch, and therefore should not be considered for hosting VMs with high communication demand across racks.

#### 6.4.1.2 Cold Spots Generation

Cold spots are constructed by continuously adding PMs to already existing collections based on several heuristics. To keep track of the non-added PMs we maintain an updated list. The first entry in the list, which corresponds to the PM with the highest rank, becomes the root of the new cold spot. The next step consists of selecting not-yet-added PMs to add to the new cold spot. Rather than simply selecting highly ranked PMs with $NRA_{PM_i}$, the algorithm also considers their distance and link utilizations to PMs that have already been added to the

cold spot. The distance is computed with the shortest-distance path algorithm proposed in (MS97), which can effectively balance the network load and utilize its resources efficiently. Let CS denote the currently identified cold spot. We define the potential of $PM_i$ as the weighted sum,

$$Potential_{PM_i} = (1 - w)R_i + w\sqrt{\frac{H_i^2 + B_i^2}{2}} \tag{6.6}$$

which includes three terms: $R_i$, $H_i$, and $B_i$.

$$R_i = \sum_{r_m \in \mathbb{R}} w_m \, ru_{i,m}$$

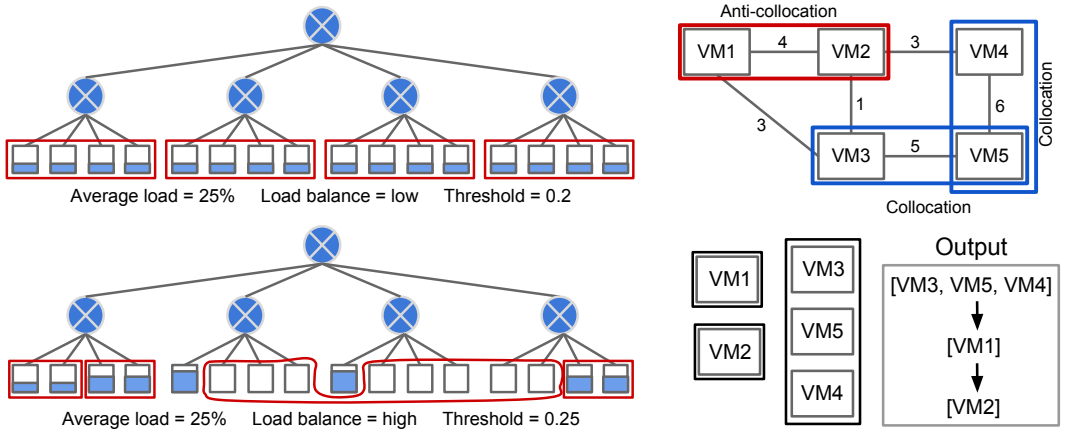
$$H_i = \frac{\sum_{PM_j \in CS} \frac{hops(PM_i,PM_j)-1}{hops(PM_i,PM_j)+1}}{n_{PM_{CS}}}, B_i = \frac{\sum_{PM_j \in CS} 1 - \frac{hops(PM_i,PM_j)}{\sum_{LK_k \in path(PM_i,PM_j)} \frac{1}{1-bu_k}}}{n_{PM_{CS}}}$$

where $R_i$ is a measure of resource utilization of $PM_i$, $H_i$ captures the distance, expressed in number of hops (hops($PM_i, PM_j$)), between $PM_i$ and all PMs that have been already added to the cold spot CS and $B_i$ the bandwidth utilization between all the links connecting $PM_i$ to all PMs in the cold spot. Both $H_i$, and $B_i$ terms are expressions of the network connectivity aspect. As such, the weight $w$ provides better controllability of the algorithm over the characteristics of the cold spot. The complexity of the algorithm is O($(n_{PM} + n_{PM_{CS}} n_{VM})\ log_{n_{VM}}$).

PMs that have lower potential values are more desirable, thus a $PM_i$ is included as part of CS if $Potential_{PM_i} \leq$ threshold. The threshold is a parameter that greatly influences the features of the resulting cold spots. Figure 6.2(a) provides an example of how cold spots are discovered for a data center consisting of 16 PMs, depending on the variance of the resources load on PMs and the threshold value. If the load variance on the PMs is low, then the prevailing factor for adding new PMs to a cold spot is their distance to PMs already found in the cold spot. In the opposite scenario, the cold spot discovery step groups PMs with similar characteristics of their compute loads and neighborhood qualities, even beyond the first-level switch. The algorithm is driven by the threshold. The lower its values, the more selective the filtering (i.e., adding PMs to a cold spot) is, and vice-versa. Thus, with a threshold of 0.2 and a low load variance, the algorithm groups together PMs under the same first-level switch.

### 6.4.2 Application Clustering

The clustering step groups the highly communicating VMs of a VNI in order to reduce traffic, while at the same time satisfying the location constraints. The purpose of performing the clustering is to guide and simplify the placement, by




(a) Cold spot discovery example (b) VNI clustering example

**Figure 6.2:** Cold spot discovery and VNI clustering examples.

establishing the order in which the VMs should be considered by the placement algorithm to improve network utilization–while respecting locality constraints.

Our proposed algorithm is based on stochastic flow injection (TC00). We extended this technique to also consider locality constraints between pairs of VMs as follows. Since anticollocation and collocation constraints translate to either placing VMs separately on different PMs or placing them together, we add logical links between location-constrained VMs with $\infty$ or $-\infty$ weights, as expressions of communication demands. Clearly $\infty$ weights correspond to collocation constraints, while $-\infty$ weights represent anticollocation. The algorithm complexity is $O(n_{LK_{CS}} {n_{VM}}^2)$, where $LK_{CS}$ corresponds to the number of links in the cold spot. Notice that other clustering techniques proposed in (MPZ10, JPE+11), such as K-mean clustering, could have been used, once extended to consider locality constraints. In the following, we describe the algorithm steps:

1. Choose a pair ($VM_i$, $VM_j$) such that the cumulative communication demand on the path is minimized and the number of hops is maximized.
2. Remove the link with lowest communication demand on the path, while satisfying the location constraints (i.e., links between VMs that have collocation constraints are not removed).
3. Repeat steps 1–2 until the VNI is disconnected or all possible ($VM_i$, $VM_j$) pairs have been considered. Clusters consist of either individual VMs or sets of VMs that have not been disconnected.
4. Perform intra-cluster ordering based on location constraints (i.e., collocation constraints precede anticollocation constraints), followed by inter-cluster decreasing ordering of VMs compute resource demands. The resulting ordering is respected by the placement algorithm when choosing VMs to place.

Note that the intra-cluster ordering is essential to respect location restrictions at placement time, while due to the inter-cluster ordering the most demanding VMs are dealt with first. This way, if resources cannot be allocated for a particular VM of a VNI, the decision to drop the VNI is made sooner. To illustrate consider a VNI composed of 5 VMs as shown in Figure 6.2(b). Assume that VM3 has the highest average compute demand followed by VM5, VM1, VM4 and VM2. The communication links between VMs have demands expressed in Mbps (e.g., 3Mbps between VM1 and VM3). Additionally, we include two collocation and one anticollocation constraints. By applying the algorithm, we generate 3 clusters of VMs, satisfying all location constraints. As expected, VM1 and VM2 need to be placed on different PMs, while all remaining VMs must be collocated – with these being hard requirements. The last step orders the VMs within each cluster based on their average compute demands, followed by a sorting between clusters. We can easily see that the first cluster considered in the placement step contains VM3, VM4 and VM5, since VM3 is the most demanding VM. Similarly, VM1 precedes VM2 at placement.

### 6.4.3 Cold Spot Selection

This step compares specific VNI features against the properties of the available cold spots and selects those cold spots that have an increased chance of allocating resources to match the VNI demands. To do this we introduce a metric $Score_{CS}$ which is used to rank all cold spots. Let us consider a VNI $\mathbf{P}$, then $Score_{CS}$ is defined as:

$$Score_{CS} = (n_{PM_{CS}} - sparsity_{\mathbf{P}}) * Avg_{CS,P} * Dev_{CS,P} \tag{6.7}$$

We describe in detail each of the three components that compose $Score_{CS}$. $Sparsity_{\boldsymbol{P}}$ provides a lower-bound in the number of PMs needed for placing $\mathbf{P}$ if all the resources in the coldspot were fully available. We explain by example how the sparsity value is computed. Consider a VNI $\mathbf{P}$ composed of $VM_1$, $VM_2$ and $VM_3$, such that their resource demands require in average 80%, 30% and 40% of a PM's capacity, respectively. Also suppose there is an anticollocation constraint between $VM_2$ and $VM_3$. This already means that in order to satisfy the constraints we need at least 2 PMs to place the VNI, because $VM_2$ and $VM_3$ need to be placed on different machines. A similar approach is followed for collocation constraints. However, when considering the VMs resource demands, placing $VM_1$ with either $VM_2$ or $VM_3$ would result in overbooking and would break our placement goals. Therefore, placing the VNI $\mathbf{P}$ requires at least 3 PMs.

$Avg_{CS,P}$ is the average remaining availability over all $n_R$ resources and the links. To define it consider cold spot $CS$ and the set of physical machines in $CS$ as $\mathbb{PM}_{\mathbb{CS}} = \{PM_i | i = 1, 2, ..., n_{PM_{CS}}\}$. Further, let $\mathbb{LK}_{\mathbb{CS}} = \{LK_k | k =$

$1, 2, ..., n_{LK_{CS}}\}$ be the set of links in $CS$. Let the VNI under consideration include the set of VMs $\mathbb{VM}_{\mathbb{P}} = \{VM_j | j = 1, 2, ..., n_{VM}\}$. We define

$$rr_m = \frac{1}{n_{PM_{CS}}}[\sum_{i=1,2,...,n_{PM_{CS}}} ra_{i,m} \; - \sum_{j=1,2,...,n_{VM}} rd_{j,m}] \tag{6.8}$$

as the average remaining availability of resource $r_m, m = 1, 2, ..., n_R$, in $CS$ after satisfying the resource demand of VNI $\mathbf{P}$.

Further, we define

$$br = \frac{1}{n_{LK_{CS}}}[\sum_{k=1,2,...,n_{LK_{CS}}} ba_k \; - \sum_{i,j=1,2,...,n_{VM};i>j} \lambda_{i,j}] \tag{6.9}$$

as the average remaining bandwidth over all links in the $CS$ after satisfying the bandwidth demand of $VNI$.

Thus, we can now define

$$Avg_{CS,P} = \frac{\sum_{m=1,2,...,n_R} rr_m \; + \; br}{n_R + 1}$$

$Dev_{CS,P}$ is the absolute deviation in remaining resource availability, given by

$$Dev_{CS,P} = \sum_{m=1,2,...,n_R} \mid rr_m - Avg_{CS,P} \mid \; + \; \mid br - Avg_{CS,P} \mid \tag{6.10}$$

In a nutshell, the first term of the equation evaluates whether the size of the cold spot is bigger than the sparsity of the VNI. The second term computes for each resource the difference between the average aggregate cold spot availability and the average aggregate VNI demand. Finally, the last term computes the overall variance we would obtain if the VNI was placed on the given cold spot – the smaller the variance, the better. The score needs to be positive for the cold spot to be considered a candidate and we always choose the cold spot with the minimum score value. The intuition behind the $Score_{CS}$ metric is that the cold spots whose features are most similar to those of the VNI should be ranked higher, and therefore tried first for placement. The algorithm complexity is $O(n_{PM_{CS}} \log n_{PM_{CS}})$ for each cold spot already discovered.

Consider the scenario from Figure 6.3, with the same VNI as in Figure 6.2(b) and four cold spots, each having 2, 4, 5 and 5 PMs, respectively. Assume that the VNI to be placed has the sparsity value 3, given by the location constraints and the fact that the PMs capacity allows neither VM1 or VM2 to be placed together with the cluster VM3, VM4, VM5. Thus, the first step of the algorithm already eliminates CS4, by comparing its size with the sparsity metric, and builds the candidates set with the remaining cold spots. Consider, in the second step, that

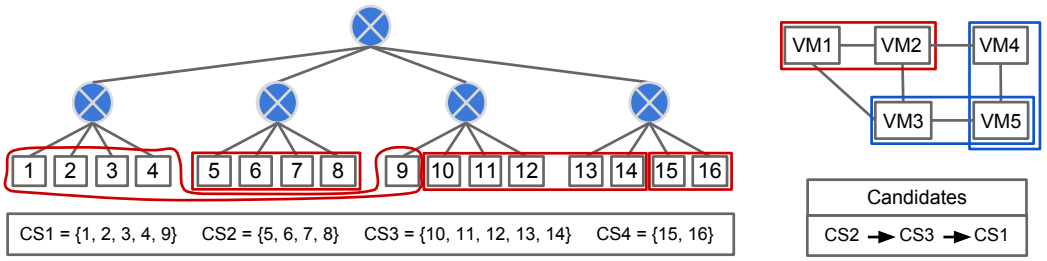



**Figure 6.3:** Cold spot selection example for clustered VNI and a data center with 16 PMs.

by computing the $Avg_{CS,P}$ and $Dev_{CS,P}$, all candidates obtain similar values and the metric differentiating them is the size against sparsity. Given the way we score the candidates, the cold spot with the smallest size (CS2) is ranked first in the placement step, while the largest ones are last.

### 6.4.4 Application Placement

The final step performs the actual VNI placement within the current cold spot, selected from the candidate list. We employ a breadth-first search algorithm, which attempts to retrieve the optimal path from the search tree based on heuristics. The search tree is an expression of the optimization problem of finding the optimal placement for a VNI. Its root is the starting search point (i.e., no VM placed yet), the inner nodes correspond to partial placements and leaf nodes to complete placements. The search tree is dynamically constructed at runtime by iteratively creating successor nodes linked to the current node. This is achieved by considering the possible placements for VMs sequentially, depending on how VMs are ordered as a result of the clustering step. A heuristic function estimating the cost of the paths from the root to the current node is used. At each step during traversal, the node with the lowest heuristic value is chosen. To exemplify such heuristics, in routing problems it can be easily expressed as the straight line distance from source to destination. In task mapping problems, it can be expressed as the minimum time of non-allocated tasks on the processor. However, for VNI placement it becomes challenging to formulate such a heuristic, since we cannot rely on notions of distance or execution time. In what follows, we discuss our heuristic used in the search algorithm given in Algorithm 5.

Our cost heuristic is an expression of the resource fragmentation in the chosen cold spot caused by the partial placement decisions, from the root to the current node in the search tree. The algorithm always advances on the path with minimum cost, by attempting to minimize resource fragmentation. Therefore, our heuristic is effectively seeking at balancing the cold spot resources utilization. Thus, for the cold spot CS, we introduce the cold spot fragmentation measure

**Algorithm 5** VNI placement algorithm.

**Input:** $VNI$ $\mathbf{P} = (\mathbb{VM}, \lambda, S)$, $\mathbb{VM} = \{\mathtt{VM}_1, ..., \mathtt{VM}_{n_{VM}}\}$, $\mathtt{CS} = \{\mathbb{PM}_{\mathbb{CS}}, \mathbb{LK}_{\mathbb{CS}}\}$, $\mathbb{PM}_{\mathbb{CS}} = \{\mathtt{PM}_1, ..., \mathtt{PM}_{n_{PM_{CS}}}\}$
**Output:** *A placement scheme* (path) *for* **P** *on* CS

1: *Initialize* pending, placed, path, V, *and* S *to* $\emptyset$
2: *For each* VM $\in$ P, *add* VM *to* pending
3: **while** pending <> $\emptyset$ **do**
4: $\mathtt{VM}_{current} \leftarrow$ pending[0]
5: **if** placed == $\emptyset$ **then**
6: *For each* PM $\in$ CS, *add* ($\mathtt{VM}_{current}$, PM, h) *to* S
7: *Add* ($\mathtt{VM}_{current}$, PM) = $min_{k \in S}\{\mathtt{h}\}$ *to* path
8: **else**
9: V $\leftarrow$ getPMsForConstraints($\mathtt{VM}_{current}$, path)
10: *For each* PM $\in$ V, *add* ($\mathtt{VM}_{current}$, PM, h) *to* S
11: *Add* ($\mathtt{VM}_{current}$, PM) = $min_{k \in S}\{\mathtt{h}\}$ *to* path
12: **end if**
13: *Remove* $\mathtt{VM}_{current}$ *from* pending
14: *Add* $\mathtt{VM}_{current}$ *to* placed
15: **end while**

denoted by $h_{CS}$, which includes contributions due to (1) *network fragmentation*, expressed as the number of isolated regions, and (2) *resource imbalance*, expressed as the deviation of utilized CPU, disk storage and memory resources within the cold spot.

We define an isolated region as a collection of PMs that share a link whose utilization is higher than 90% when communicating to the rest of the network starting from the first level switch. To illustrate, all the PMs contained in a bladecenter whose link connecting to the rack-level switch is 92% utilized would comprise an isolated region. Let $N_{isolatedregions}$ be the number of isolated regions in $CS$. In order to compute its value, we implemented a recursive algorithm that follows a top-down approach. Starting from the data center root node, it walks the adjacent core links, such that if their utilizations surpass the 90% threshold, the switches and corresponding PMs below them result in different isolated regions. At every step, the algorithm descends an additional level in the data center hierarchy, until it reaches the first level switches.

As described earlier, the cold spot CS consists of the set of physical machines $\mathbb{PM}_{\mathbb{CS}} = \{PM_i | i = 1, 2, ..., n_{PM_{CS}}\}$. We define $ra_m = \frac{1}{n_{PM_{CS}}} \sum_{i=1,2,...,n_{PM_{CS}}} ra_{i,m}$ as the average availability of resource $r_m, m = 1, 2, ..., n_R$ in CS. Further, we

define

$$Avg_{CS} = \frac{1}{n_R} \sum_{m=1,2,...,n_R} ra_m, Dev_{CS} = \sum_{m=1,2,...,n_R} \mid ra_m - Avg_{CS} \mid$$

as the average availability over all $n_R$ resources in CS and the absolute deviation in resource availability, respectively.

We denote the cold spot fragmentation measure as

$$h_{CS} = \frac{N_{isolatedregions} * Dev_{CS}/n_R}{Avail_{CS}}, \tag{6.11}$$

where $Avail_{CS}$ denotes the overall availability of $CS$ and is given by

$$Avail_{CS} = \frac{\sum_{i,j=1,2,...,n_{PM_{CS}};i>j} \frac{(ra_{PM_i}+ra_{PM_j})}{2} * min_{k \in path(PM_i,PMj)}\, ba_k}{n_{PM_{CS}}\,(n_{PM_{CS}} - 1)} \tag{6.12}$$

where

$$ra_{PM_i} = \frac{1}{n_R} \sum_{m=1,2,...,n_R} ra_{i,m} \tag{6.13}$$

The algorithm complexity is $O(n_{PM_{CS}} + {n_{VM}}^2)$. In order to speedup the VNI placement, we consider a simple, but effective variant based on beam search. Instead of accounting for all valid PMs for the current VM, by checking that location constraints relative to already placed VMs are satisfied, only a reduced number of PMs are processed. Given the previously placed VMs, for the current VM we consider those PMs that are closest in the number of hops to all the PMs already allocated. Only if, by computing the heuristic, none of the closest PMs have the necessary resources for the current VM, we expand the search by including the PMs that have not been considered in the first step. In most cases, the solution found by applying beam search will be suboptimal, but significantly faster.

**Placement improvements.** Currently, if neither of the candidate cold spots can successfully allocate resources for the VNI, the placement request is dropped. In order to reduce the drop rate we consider several improvements. First, merging cold spots is a viable approach to producing cold spots with more PMs and higher resource availabilities, such that more demanding or constrained VNIs can be placed. Second, the placement algorithm could be improved by using A* search (HNR68). This means that partial placement decisions are based on the current resource fragmentation, as well as estimations of the final resource fragmentation. Such an enhancement can potentially prune further paths in the search space and achieve better placement outcomes. A third improvement refers to softening the location constraints. Although this changes our placement goals,

it may be preferable to provide users with modified placement schemes than originally intended, instead of dropping their requests. In the first phase, one can soften the collocation constraints, such that VMs are placed as close to each other as possible (i.e., on PMs located under the same first level switch) or on PMs that share high available links, if the collocation constraint results from a high communication demand. In the second phase, VMs that are anticollocation constrained can be placed on the same PM. The reason for softening collocation before anticollocation is related to what each constraint offers at application level. While, collocation constraints mainly impact the application response time, anticollocation affects the application robustness.

## 6.5 Experimental Evaluation

Further, we present simulation results to demonstrate the performance of our VNI placement technique. The method of batch means is used to estimate the performance parameters considered, with each batch consisting of 15 runs. For every run, the following methodology is used. We start with an empty data center and sequentially place VNIs until its average compute load – as mean over CPU, memory and disk storage usages – reaches 25%, 50% and 75%. Next, we remove random placed VNIs and add new ones with an exponential distribution, such that the average compute load remains stable around the respective targeted values. Each experiment is run such that 50% of the initially placed VNIs are replaced by new VNIs and we collect the performance metrics periodically. Our simulator was written in Java and the experiments were performed on a ThinkPad T520, with 4GB RAM and Intel Core i3-2350m processor, running Ubuntu 11.04.

We consider three type of performance metrics which capture the perspective of the *VNI*, *user* and the *system*. The average path length per placed VNI represents the VNI metric and captures the distance in number of hops between VMs belonging to the same VNI instance. We consider three user metrics: (1) placement time, which represents the time it takes to solve the placement problem, (2) number of attempts, which captures the average number of attempts or cold spots considered until successfully placing a VNI, and (3) drop rate, which is a ratio of the number of rejected VNIs over the total number of offered VNIs. Finally, the system metrics include the (1) average network utilization, (2) average network congestion, defined as the mean over most congested links per placed VNI, (3) network variance, and (4) compute resources variance.

Tree networks are widely adopted in data centers due to their cost-effectiveness and simplicity. Therefore, we consider a data center consisting of a three level tree structure. Following a bottom-top order it can be described as follows: the first level consists of PMs, the second level consists of bladecenters – with each bladecenter containing 16 PMs, the third level consists of racks – with each

rack containing 4 bladecenters. We vary the number of racks to produce data centers of different capacities, where by capacity we refer to the size of the data center in number of PMs. We consider three data center sizes: 64, 256 and 1024 PMs. Each PM has 32 cores, 64GB RAM and 4TB storage capacities, while each network link has 1Gbps capacity. Additionally, between any two PMs there exists a unique path in the data center. Following, the data center diameter defined as the maximum number of hops between any two PMs is 4 for 64 PMs, 6 for 256 PMs and 8 for 1024 PMs. We note here that our techniques can be easily applied to other data center topologies, such as VL2 (GHJ+09), Bcube (GLL+09) or Portland (MPF+09).

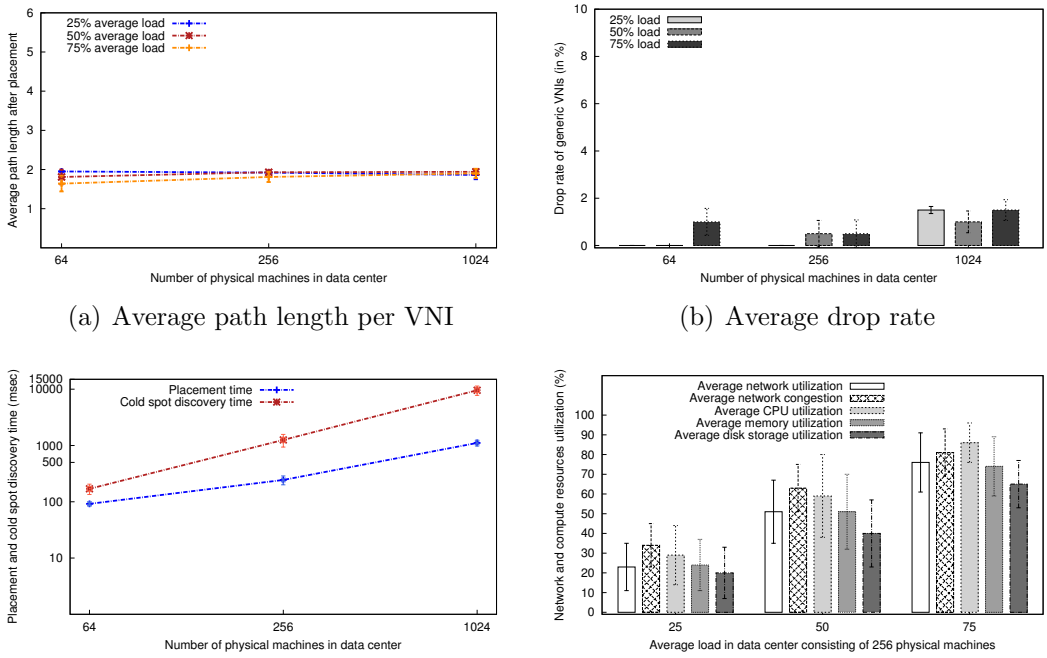
A rich set of workloads is considered. We first evaluate our technique against a generic VNI mix in Chapter 6.5.1 and show the impact that each placement stage has on the performance of our approach. Second, we consider a more realistic workload mix consisting of cache, hadoop and three-tiered like VNIs as described in Chapter 6.5.3. Finally, we report on the impact that the threshold value has on the cold spot discovery step (Chapter 6.5.4) and compare our approach with a technique proposed for virtual network embedding (ZA06) in Chapter 6.5.5.

### 6.5.1 Generic Application Mix

We consider three types of VNIs, namely small, large and extra-large consisting of small, large and extra-large VM instances, respectively. The resource demands of the VMs follow the specifications of Amazon EC2 instances (EC2). That is, their respective resource demands are: (1 core, 1.7GB memory, 160GB storage), (4 cores, 7.5GB memory, 850 GB storage) and (8 cores, 15GB memory, 1690 GB storage). A generic mix is composed of 60% small VNIs, 25% large VNIs and 15% extra-large VNIs. The number of VMs per VNI is between 2 and 10 following a uniform distribution. For every pair of VMs, we create network demand and locality constraints with probability 0.5 and 0.1, respectively, with the ratio of collocation to anticollocation constraints being 0.5. The network demands between small, large and extra-large pairs of VMs are 5, 20 and 50 Mbps, respectively.

Results are shown in Figure 6.4. As it can be observed, the average path length for placed VNIs remains stable at a value of 2 hops and is independent of the data center diameter, thus demonstrating the scalable nature of our approach. This is due to the fact that cold spots enable keeping network traffic under the first-level switch, hence reducing network traffic across higher-level switches. We also note that the average number of attempts to place a VNI varies between 1 and 2, which demonstrates that our selection techniques is effective at ranking cold spots as a function of how their features compare to the offered VNI.

Figure 6.4(b) depicts a low drop rate of less than 2%. More specifically, for the 64-PM, 256-PM and 1025-PM data center, (62, 118 and 184), (239, 468 and 723) and (977, 1858 and 2820) VNIs are offered in total, respectively for the 25%, 50%




(a) Average path length per VNI

(b) Average drop rate

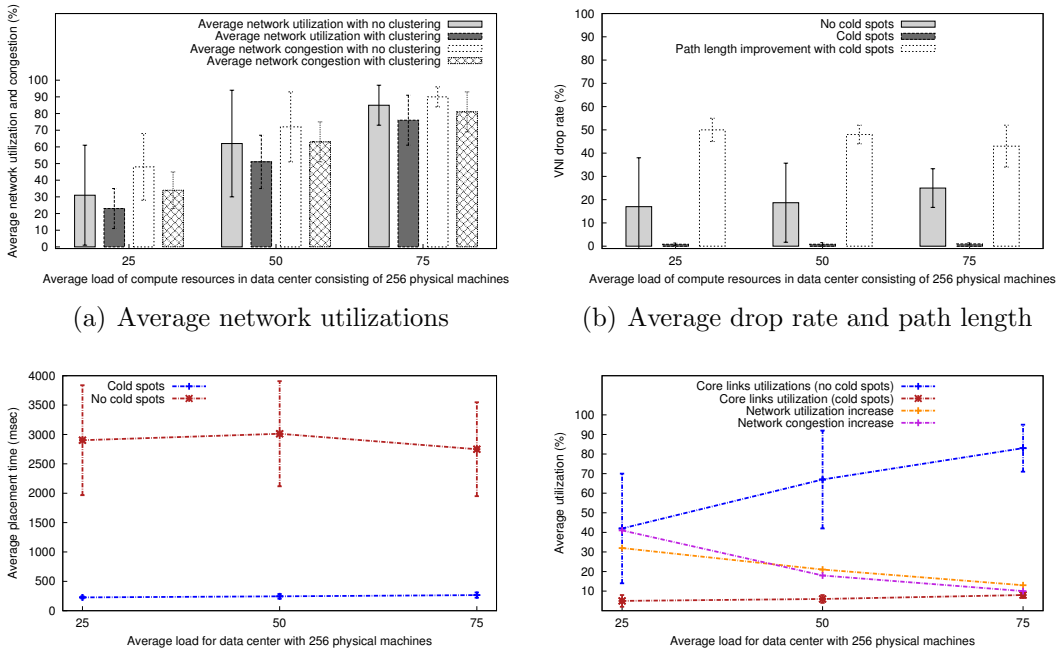
(c) Placement and discovery times

(d) Network and compute utilizations

**Figure 6.4:** Results for the generic VNI workload with various data center sizes.

and 75% loads. As expected, we observe in Figure 6.4(c) that the placement time increases linearly with the size of the data center, going from 90ms for 64 PMs to 1100ms for 1024 PMs. Similarly, the cold spot discovery time increases as the data center becomes larger, from ≈170ms for 64 PMs, to ≈1255ms for 256 PMs, and to ≈9590ms for 1024 PMs. However, recall that this time is amortized since the cold spot discovery step is executed asynchronously to actual VNI placement calls.

Figure 6.4(d) considers the 256-PM data center and shows the average compute and network utilizations, as well as their corresponding variances, as measures of resource load balancing. Note that Amazon EC2 instances are CPU intensive, therefore the CPU load for all three loads (i.e., 25%, 50%, 75%) is slightly higher than memory and disk storage. Although, the standard deviation is higher when the data center load reaches 50%, its value rarely reaches 20%. To characterize the data center network, we measure the average utilization and the average congestion. The network utilization has similar values as the compute one and its deviation is less than 16%. As expected, the network congestion is higher than the utilization, since for every placed VNI it accounts only for the most congested link on the path between the corresponding PMs. We observe that the maximum network congestion VNIs experience is 81% corresponding to




(a) Average network utilizations

(b) Average drop rate and path length

(c) Average placement time

(d) Average network performance

**Figure 6.5:** Network performance with vs. without VNI clustering (a) and cold spot vs. data center level (b-d) in a 256 PMs data center.

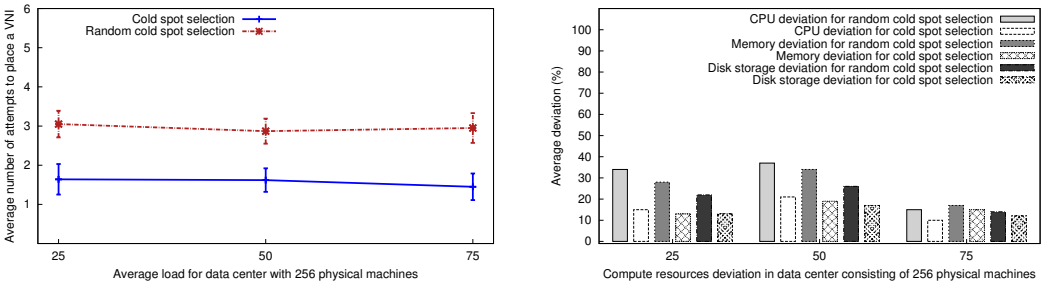
a load of 75%.

### 6.5.2 Breaking Down the Placement Technique

In this section we investigate the impact that each individual placement step has on the overall performance. To do this we repeat the experiments with each individual step disabled or modified as described below.

**VNI clustering**. We first repeat the experiment for the 256 PMS data center with the VNI clustering step disabled. As it can be observed in Figure 6.5(a), without clustering the network utilization and congestion increases by 10–25% and 10–30%, respectively. Furthermore, the variance in the links utilization is higher by up to 60% for lower data center loads, while the congestion variance is higher by up to 45%. It can be concluded the clustering step is effective at ensuring that highly-communicating VMs are placed in close proximity, which, as a result, improves network utilization and load balancing. However, we note that for cache and hadoop VNIs the clustering step does not provide similar improvements. This is due to the fact that all VMs have the same node degrees and demands, thus making the clustering step less effective.

**Cold spot discovery**. Following, our previous results for the generic VNI work-




(a) Average attempts per VNI

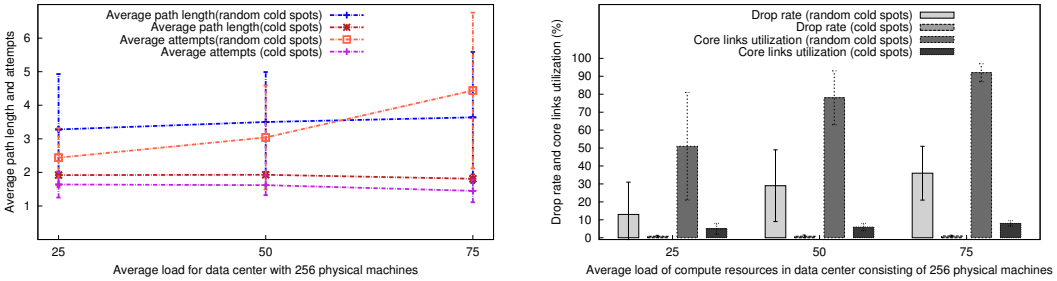
(b) Average compute variance

**Figure 6.6:** Random cold spot selection vs. our default algorithm in a 256 PMs data center.

load mix are compared with those obtained when disabling the cold spot discovery step. That is, for every incoming VNI, the placement consider all the PMs in the data center for placement. Results are shown in Figures 6.5(b)–6.5(d). We observe that without cold spots the average path length for placed VNIs increases by a factor of 2. That is, VMs belonging to the same VNI are placed further apart from each other, thus impacting the traffic in the core links. In fact, we observe the core links utilization increases up to 10x factor. As a consequence of this, the drop rate increases from less than 2% to up to 25%. This is due to the fact that as core links become congested, the network becomes fragmented and it is more difficult to find a feasible placement for incoming VNIs. A secondary effect is observed in the increased average network utilization and congestion by up to 40% for lower loads of the data center. Finally, given that to place a VNI the algorithm considers all the PMs in the data center, the placement time increases by 10–12x factor as shown in Figure 6.5(c).

**Random cold spot selection**. Further, we are interested in assessing our cold spot selection technique. To do this let us consider a selection algorithm wherein cold spots are selected in a random fashion. Figure 6.6 shows the average placement attempts and variance of compute resources. As observed, randomly selecting cold spots increases the number of attempts required for successful placements by a factor of 1.5–2x. In addition, the compute resources variance is higher by 15% to 55%, with the more significant impact for lower data center loads. Similarly, the network utilization and congestion are also increased by up to 40% and 15%, respectively. We conclude that the cold spot selection step intelligently chooses the best candidate cold spots for each VNI, to achieve better load balancing and VNI performance in the data center.

**Random cold spot discovery**. Finally, we evaluate how our cold spot discovery technique influences the performance of our placement technique (Figure 6.7). Let us consider a cold spot discovery algorithm wherein PMs are randomly added to




(a) Path length and attempts per VNI

(b) Drop rate and core links utilization

**Figure 6.7:** Comparison between random cold spots and our default algorithm on 256 PMs data center.

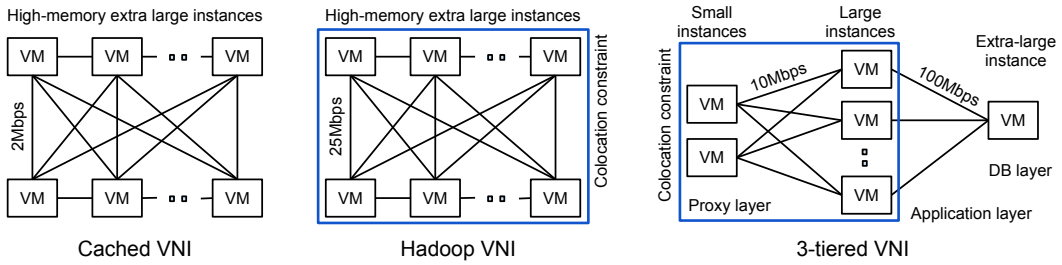


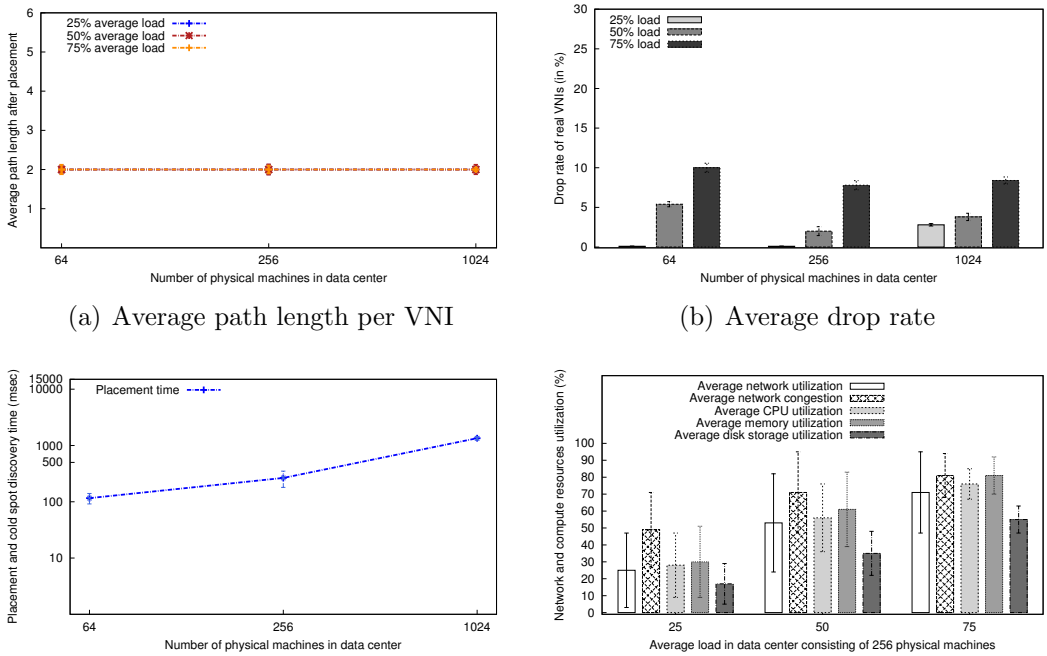

**Figure 6.8:** Topologies for the cache, hadoop and three-tiered VNIs.

cold spots, as opposed to being added based on their rankings. In this algorithm, the size of the randomly generated cold spots corresponds to the average observed in our previous experiments which is 16. This step is invoked every 20 new incoming VNIs. As expected, the average path length of the placed VNIs increases to 3–4 hops and in some cases even reaches the data center diameter. We also observe an increase in the average number of attempts to place VNIs. Given the random locality in the data center of the VMs within one cold spot, many VNI placements impose demands on the core links. As such, we notice an increase to up to 90% utilization, as opposed to utilizations under 10% with our technique. An important effect of the core links congestion is the increased drop rate, to up to 36% of the total number of offered VNIs.

### 6.5.3 Cache, Hadoop and Three-tiered Applications

The second part of the evaluation considers realistic workloads, composed of cache, hadoop and three-tiered -like VNIs, and measures the effectiveness of our placement technique for the performance metrics considered in the previous section. Figure 6.8 shows the topologies corresponding to these specific VNIs.

Note that cache and hadoop VNIs have identical topologies, where all VMs




(a) Average path length per VNI

(b) Average drop rate

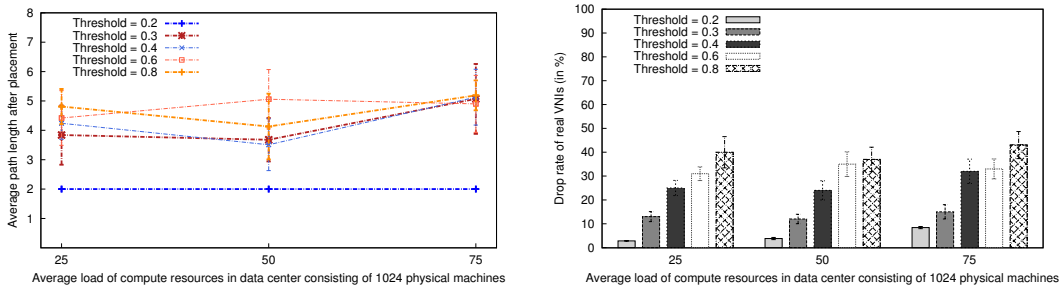
(c) Average placement time

(d) Network and compute utilizations

**Figure 6.9:** Results for mix of cache, hadoop and three-tiered VNIs with various data center sizes.

communicate in a full mesh model and their number varies between 6 and 12. The compute demands match the specifications of Amazon EC2's high-memory extra large instances (6.5 cores, 17.1GB memory, 420GB storage). The network demands are 2Mbps and 25Mbps for the cache and hadoop VNIs, respectively. Whereas, the cache VNI has no location constraints, the VMs of hadoop VNIs need to be placed on PMs located under the same first-level switch (bladecenter). The three-tiered VNIs contain a proxy layer with 2 small EC2-like instances, an application layer, consisting of 5 to 10 large EC2-like instances, and the database layer with 1 extra-large EC2-like instance. The network connectivity between layers is full mesh, with 10Mbps demand for proxies and 100 Mbps for the database instance. The location constraints apply to the VMs belonging to the application and proxy layers, such that they need to be placed on PMs located under the same first-level switch.

Figure 6.9 reports the results obtained when placing a mix of realistic VNIs, where 50% are three-tiered, 25% are cache and the remaining 25% are hadoop. The number of VNIs offered for the 256 PMs data center is 70, 144 and 199 VNIs, corresponding to 25%, 50% and 75% average loads. Similarly, in the 1024 PMs data center, 281, 573 and 859 VNIs are placed. We observe that the average path




(a) Average path length per VNI

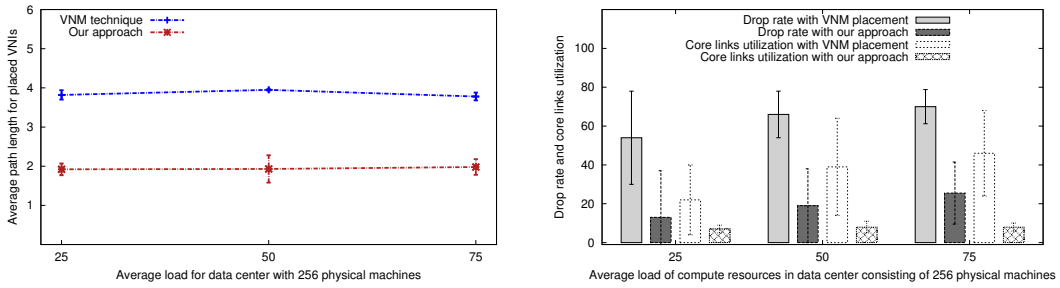
(b) Average drop rate

**Figure 6.10:** Results with different threshold values in a 1024 PMs data center.

length per placed VNI is 2 and remains independent of the data center diameter. It is also noticeable that the network constrained nature of the cache and hadoop VNIs results in higher drop rate of up to 8% when the average load in the data center is 75% and longer placement time by at most 20% as compared to placing generic VNIs. This results in an increase in the number of attempts to place a VNI to an average between 2 and 3.2. In Figure 6.9(d) we note that the average memory load is higher than both CPU and disk storage. This effect is due to the higher memory footprint of both cache and hadoop VNIs. Second, when looking at network utilization, we notice a similarity with the results obtained for generic VNIs. This is explained by the cheap network demands of cache VNIs in particular, which compensate for the higher number of communication links between VMs. We also observe that the network congestion increases by up to 30% for lower loads as compared to when generic VNIs are used due to the higher network demand of the VNIs.

### 6.5.4 Cold Spot Discovery Threshold

Next, we investigate how the threshold used in the cold spot discovery step influences the properties of the cold spots and the effectiveness of our technique. To do this the simulation is run with the mix of realistic workloads on a 1024 PMs sized data center for various threshold values. Selective results are shown in Figure 6.10. We notice an increased in average path length per VNI placed as the threshold increases. Over extensive experiments, we found that a value of 0.2 results in cold spots that are balanced and PMs are closely located. With other values, one can easily generate cold spots that are either too small in size, and thus not suitable for the offered VNIs, or too large, and thus making the placement process less effective. As a result, the drop rate increases with higher thresholds, reaching 43% in some cases. This follows intuition since VMs are located further from each other, which means core links quickly become congested.




(a) Average path length per VNI

(b) Average drop rate

**Figure 6.11:** Comparison between our approach and the VNM technique proposed in (ZA06) for a 256 PMs data center.

Clearly choosing different thresholds can impact the performance of our technique. Following, tuning the threshold parameter represents a complex problem and depends on factors such as the topology and resource demands of the incoming VNIs or the current average load and load variance in the data center. We discuss further in Chapter 6.6.

### 6.5.5 Comparison to Virtual Network Mapping

Finally, we want to compare our approach with previous techniques.The closest work to ours is a virtual network mapping (VNM) technique previously proposed in (ZA06). In (ZA06), the authors recognize the need to consider both physical node and links optimizations together throughout the placement process. Additionally, the algorithm controls the network allocation (routing) and therefore has more flexibility at finding a feasible placement. The algorithm in (ZA06) finds a cluster of physical nodes that are lightly loaded without considering the network connectivity of the physical nodes and solves the routing problem of connecting physical nodes based on the topology of the virtual network. To match the authors' assumption that one physical node can only allocate resources for one virtual node, we apply our placement technique on generic VNIs whose VMs and links require at least 50% of a PM's compute resources and bandwidth capacity. We note that this is an unrealistic assumption since in practice, VM to PM densities as high as 100 VMs per PMs are the norm in cloud environments. However, it allows us to perform a more fair comparison between both techniques.

The results obtained for a 256 PMs data center are presented in Figure 6.11. As expected, satisfying the compute constraint of 50% demand imposes additional difficulty on our algorithm, which results in an increased drop rate to up to 25%. However, since the VNM technique does not consider the ample network connectivity of physical nodes, the resources chosen do not properly match the

nature of VNIs. Consequently, the drop rate increases to up to 70% and the core links utilization increases as the average load of the data center increases. Similarly, the average path length per placed VNI is higher by a 2x factor. We conclude that considering locality and neighborhood quality in the cold spot discovery are primary factors for resource allocation in data centers.

## 6.6 Summary and Discussion

In this chapter, we have considered the problem of placing virtual networked infrastructures with compute, network and availability constraints in large data centers. Unlike previous approaches that address the placement problem from either the network or computer resources perspective, our approach factors in both in one integrated solution. Our placement framework makes the problem tractable and is generic enough to support increasing complexity. The center of the proposed technique lies in the introduction of cold spots, defined as resource constructs that reduce the combinatorial complexity of the problem, while enabling the load balancing of resources. Thus, we break the problem in four steps: (a) identifying colds spots, (b) clustering virtual machines to reduce overall communication traffic and reduce placement complexity, (c) identifying candidate cold posts whose features are similar to those of the VNI in order to increase the chances of deployment, and (d) performing the actual placement by using efficient graph-based search algorithms that optimize for load balancing. Extensive simulations show the effectiveness and efficiency of our approach with a rich set of workloads, such as *hadoop-* , *cache* and *3-tier-* like infrastructures. The drop rate is under 2% for generic VNIs and under 8% for the more realistic workloads, while in the same a good load balancing is achieved.

Further, we address some additional considerations towards a more complete application placement framework to be considered as future work.

*Application-aware dynamic cold spot discovery.* We have shown that constructing cold spots based on network and compute resource availability suffices to achieve good placement performance for workloads with compute, network and availability constraints. As workloads become more complex, the process of cold spot discovery need to be extended to address workloads characteristics, such as proximity to specific storage devices or maximum acceptable latency between specific virtual machines. We favor an online approach wherein via learning mechanisms the cold spot discovery process is tuned to identify cold spots whose properties are aligned with the incoming workload.

*Integration into a real system.* Our technique is model driven, therefore to adopt it in a real environment one requires to build a model of the data center and the workloads. This is a simple software engineer task and in fact, significant part of the placement technique presented has been deployed in a data center

environment.

*Optimizations.* First, we foresee being able to merge and split cold spots in order to produce cold spots with specific characteristics for different workloads, as well as to increase even further the acceptance ratio for the more demanding ones. Second, when such applications leave the system, we do not keep track of their placement scheme. This could be used to improve the placement efficiency of future applications, by comparing their characteristics with those from already placed ones. Third, the placement algorithm could be improved by using A* search so that partial placement decisions are based on estimations of the final resource fragmentation. This enhancement has the potential of pruning paths in the search space even further and achieving better placement outcomes. Fourth, both the cold spot discovery and placement algorithms can be further engineered to improve their execution time.

# Chapter 7

# Conclusions

Current trends in computing provide users with increased processing power and storage, available in cloud data centers. In the same time, modern devices feature a greater ease of connectivity. For instance, the explosion in popularity of mobile devices has determined the quick advancement of cellular wireless networks from 1G to 4G, thus making mobile communications faster than ever. Energy efficiency has also become significantly relevant and most providers are proactively trying to reduce power consumption. To benefit from these trends, users demand richer, more complex applications and still want to maintain the same fast and spontaneous interactions. Such expectations create a sophisticated design problem that can no longer be addressed by running these applications on single machines. In the context of mobile cloud computing, it means either executing the application on the mobile device or in the cloud, while the device acts as a thin client. Unfortunately, these traditional approaches lack the flexibility to provide users with spontaneous and shorter interactions.

This thesis contributes towards overcoming such limitations by proposing an alternative approach to integrate cloud-based applications with mobile devices. To achieve this, we exploit the unexplored space between these two extreme alternatives, such that parts of the application computing process are outsourced from the cloud to the client side. The thesis is concerned with improving user experience, by finding the balance between the improvements in application performance offered to users, extending the battery life on the mobile devices and how resources are utilized on both the client and cloud sides. These aspects are addressed throughout three main parts.

First, we propose a modular-based model that flexibly and optimally distributes cloud-based applications on mobile devices and virtual instances, such that interaction time is minimized and battery life is extended. This is achieved by solving an optimization problem that attempts to minimize data transfers and the execution time of computational steps. Unlike previous work, we remove the assumptions that applications are pre-installed on the mobile device and that

their data is always presents at both ends. Instead, the focus is on the challenging aspect of mobility, where optimally distributing such applications must consider the costs incurred by on-the-fly data and code migration. The model uses modularization to provide the basis for flexible distributed deployments, from having only the user interface to hosting the entire application on the mobile device. Our approach, implemented in the AlfredO framework, dynamically distributes application modules and adjusts the application deployment to optimize its behavior and performance. The dynamic aspects guarantee that on-the-fly application acquisition does not result in unacceptable delays and that the system can adapt to the changing conditions one is likely to encounter when operating with mobile devices, such as CPU load, network bandwidth and latency or data loads. Extensive experiments with various modular applications show that our approach significantly improves performance and reduces power consumption on the mobile device. Thus, the system takes an important step towards a better utilization of mobile resources and an optimal deployment of cloud-based applications.

The second part of the thesis looks at modeling the performance estimation of such applications to answer a more general question: What is the optimal (or nearly optimal) distribution of an application when certain workloads and specific mobile device – cloud virtual instance environments are considered? This problem is motivated by the lack of practical feasibility to gather statistics for all applications by considering all valid distribution schemes and execution environments. Such extensive measurements are required by AlfredO to discover the partitioning with the lowest interaction time. Therefore, we propose a queuing network-based model that is able to estimate the optimal distributions, while requiring only passive measurements. The model combines in a unified solution important factors that impact an application's performance – valid distribution schemes, workload, as well as the resource capabilities and usages of the considered mobile device and cloud virtual instance. It understands what are the application demands for specific resources and how specific execution environments vary in their resource availabilities from those for which passive measurements exist. Workloads are modeled as request-reply nonstationary interactions and regression approaches on gathered statistics are used to obtain an accurate model. Evaluation on various scenarios shows that our approach chooses the optimal or nearly optimal distribution scheme and significantly improves application performance over using AlfredO with passive measurements only.

The third part of the thesis switches the focus from the mobile device to how to efficiently distribute the cloud-located application modules in large data centers. Such applications consist of collections of heterogeneous virtual machines connected via pair-wise connectivity models. Specific constraints, such as high-availability and high-communication, govern their performance as a whole in order to satisfy different requirements. The former allows the cloud to contain the impact of failure-locality, while the latter allows it to provide good applica-

tion performance to users. The fundamental challenge is the NP-hardness of the problem, due to its combinatorial nature. Therefore, the primary goal is efficiency together with achieving a high application acceptance rate and a good balancing of resources. Instead of addressing this distribution problem in a piecemeal fashion, our approach is comprehensive in that it factors the network, compute and availability performance aspects into one single solution. The main focus is the introduction of a resource abstraction, namely cold spots. A cold spot consists of a collection of physical computing nodes that exhibit high availability of compute resources and network connectivity. By using them for distribution, they reduce the search space of the optimization problem. Simulations with a rich set of applications show that our technique is efficient and scalable, provides good load balance in the data center and achieves a high application acceptance rate.

The three parts occupy different points in the context of integrating cloud-based applications with mobile devices and tackle challenges from both the client and cloud perspectives. Neither of the contributions claims to be the most effective or the only possible approach to the problem it addresses. For instance, some of the challenges in AlfredO could potentially be simplified or avoided by choosing a different application model than OSGi modules. Previous work has proposed Java methods or classes as the offloading code unit. Although these alternatives are efficient for small applications, in the case of large applications the search space for optimization increases exponentially. OSGi was chosen because well-designed modular decompositions of applications together with a runtime system able to actively manage their deployments, offers the flexibility of dealing with dynamic environments and the proper granularity to make the optimization problem tractable. In practice, quite few applications are properly modularized. Our experience from either building modular applications from scratch or modularizing existing ones, shows that the latter can become an important challenge especially for large code bases with poor separation of concerns. For the performance estimation model, alternatives to queuing networks, such as Petri nets or probabilistic formulations, could have been considered. Given the nature of modular applications, queuing networks were chosen because of their ability to naturally emulate the behavior and dependency model of such applications. Moreover, a queue-based model can potentially simplify future enhancements such as modeling multiple resources at module level, supporting multiple clients or application idiosyncrasies, like session-based workloads, module replication and load imbalances across replicas.

## 7.1 Future Work

A possible direction for future work is enhancing AlfredO and the performance estimation model to support parallelism and concurrency for newly emerging classes

of applications. For instance, interactive perception applications use high-data rate sensors to perform face or object recognition and enable augmented-reality experiences on mobile devices. Such applications require crisp responses and continuous processing of high data rate sensors to maintain accuracy, thus they can benefit from parallelism on multi-core systems. From the development perspective, writing applications with the parallel hardware aspect in mind is challenging. Modular designs can potentially deal with the hardware challenges and support building highly concurrent and parallel applications. Perhaps the most eloquent scenario is the one where multiple mobile users simultaneously interact with the same application hosted in the cloud, and therefore with the same software modules. The resulting parallelism can be exploited through self-adaptive systems, where AlfredO enhanced with concurrency support could optimize the distributions of applications for these users. Additionally, data availability is relevant for completing user interactions with a running application. An important challenge is understanding the trade-offs between data transfer strategies and data persistence, when network connectivity, bandwidth, mobile device capacity and latency are considered.

Another research focus in the context of application distribution in the cloud is the dynamic discovery of cold spots. The motivation is the increasing complexity and specialization of applications currently deployed. As already seen with the realistic workloads considered, applications can predominantly demand memory resources, processing power, higher network bandwidth or proximity to specific storage volumes. Therefore, an efficient distribution can benefit from the dynamic discovery of cold spots, wherein cold spots are specialized such that their properties are aligned with the incoming applications. Moreover, this idea can be naturally integrated with the previously mentioned direction, of adding support for concurrency and parallelism. We can easily envision applications spread over several virtual machines and simultaneously used by multiple mobile users, for which improving deployment triggers better experiences.

Finally, another interesting direction is looking at the dynamic aspects of application distribution, by considering reconfiguration. In the process of resource allocation, network conditions change over time dues to the arrival and departure of applications, and thus reconfiguring placed applications can improve performance and resource utilization. However, periodic reconfigurations are not desirable for several reasons. First, the process of reconfiguration is expensive, since it includes both the cost of recomputing deployments and the service disruption cost. Second, the reconfiguration order of different applications can potentially affect the performance of other applications. Therefore, the challenges are to identify hot spots in data centers, namely those collections of physical machines and links that are highly utilized, and to reconfigure only the most critical applications allocated, such that acceptable load balancing is achieved.

# Bibliography


[AG01] Fernando Brito Abreu and Miguel Goulao. Coupling and Cohesion as Modularization Drivers: Are We Being Over-persuaded? In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, pages 47–57, 2001. 43

[And07] Android, 2007. code.google.com/android. Retrieved on august 2010. 29, 39, 42

[Apa10] Apache Felix, 2010. felix.apache.org/site/index.html. Retrieved on August 2010. 39, 42

[APGG00] Khalil Amiri, David Petrou, Gregory Ganger, and Garth Gibson. Dynamic Function Placement for Data-intensive Cluster Computing. In *Proceedings of the 18th USENIX Annual Technical Conference (USENIX'00)*, pages 307–322, 2000. 11

[App11] Google AppInventor, 2011. appinventor.googlelabs.com/about. 39

[ASK06] Tarun Agarwal, Amit Sharma, and Laxmikant V. Kale. Topology-aware Task Mapping for Reducing Communication Contention on Large Parallel Machines. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*, pages 145–145, 2006. 20

[AW96] Martin F. Arlitt and Carey L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'96)*, pages 126–137, 1996. 17

[Bal06] Rajesh K. Balan. Simplifying Cyber Foraging. *PhD Thesis, School of Computer Science, Carnegie Mellon University*, 2006. 2

[BASS11] Theophilus Benson, Aditya Akella, Anees Shaikh, and Sambit Sahu. CloudNaaS: A Cloud Networking Platform for Enterprise Applica-

tions. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC'11)*, pages 1–13. ACM, 2011. 21, 122

[BBV09] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy Cosumption in Mobile Devices: A Measurement Study and Implications for Network Applications. In *Proceedings of the 9th Internet Measurement Conference (IMC'09)*, 2009. 45

[BCB10] Nabeel Farooq Butt, Mosharaf Kabir Chowdhury, and Raouf Boutaba. Topology-awareness and Reoptimization Mechanism for Virtual Network Embedding. In *Proceedings of the 9th International Conference on Networking (NETWORKING'10)*, volume 6091, pages 27–39, 2010. 20

[BD11] Fabian Beck and Stephan Diehl. On the Congruence of Modularity and Code Coupling. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11)*, pages 354–364, 2011. 43

[BDIM04] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for Request Extraction and Workload Modeling. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI'04)*, pages 18–31, 2004. 18, 93

[BDR+07] Erik Boman, Karen Devine, Sivasankaran Rajamanickam, Umit Catalyurek, and Doruk Bozdag. Zoltan: Parallel Partitioning, Load Balancing and Data-Management Services User's Guide. *Sandia National Laboratories*, 2007. 57

[Ben02] Endika Bengoetxea. Inexact Graph Matching Using Estimation Distribution Algorithms. *Phd Thesis, Ecole Nationale Superieure des Telecommunications Paris*, 2002. 122

[BFS+02] Rajesh Balan, Jason Flinn, Mahadev Satyanarayanan, Shafeeq Sinnamohideen, and Hen I. Yang. The Case for Cyber Foraging. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop: Beyond the PC (EW'02)*, pages 87–92. ACM, 2002. 10

[BM05] Mohamed N. Benani and Daniel A. Menasce. Resource Allocation for Autonomic Data Centers Using Analytic Performance Models. In *Proceedings of the 2nd International Conference on Automatic Computing (ICAC'05)*, pages 229–240, 2005. 16

[BR73] Ian Barrodale and Frank Roberts. An Improved Algorithm for Discrete L1 Linear Approximations. In *SIAM Journal of Numerical Analysis*, volume 10, pages 839–848, 1973. 110

[BRS99] Hari Balakrishnan, Hariharan S. Rahul, and Srinivasan Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *Proceedings of ACM Conference on Applications, Technologies, Architectures and Protocols for Computer Communication (SIGCOMM'99)*, pages 175–187. ACM, 1999. 53

[BSPO03] Rajesh Balan, Mahadev Satyanarayanan, So Y. Park, and Tadashi Okoshi. Tactics-based Remote Execution for Mobile Computing. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services (MobiSys'03)*, pages 273–286. ACM, 2003. 10, 38

[BSSH05] Matthew Britton, Venus Shum, Lionel Sacks, and Hamed Haddadi. A Biologically-inspired Approach to Designing Wireless Sensor Networks. In *Proceedings of the 2nd European Workshop on Wireless Sensor Networks (EWSN'05)*, pages 256–266, 2005. 21

[BZC06] John J. Barton, Shumin Zhai, and Steve B. Cousins. Mobile Phones Will Become the Primary Personal Computing Devices. In *Proceedings of the 7th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'06)*, pages 3–9, 2006. 37

[Cas06] Giuliano Casale. An Efficient Algorithm for the Exact Analysis of Multiclass Queuing Networks With Large Population Sizes. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance'06)*, pages 169–180, 2006. 15

[CBC+10] Eduardo Cuervo, Aruna Balasubramanian, Dae-Ki Cho, Alec Wolman, Stefan Saroiu, Raveer Chandra, and Paramvir Bahl. MAUI: Making Smartphones Last Longer With Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications and Services (MobiSys'10)*, pages 49–62. ACM, 2010. 12, 38, 51

[CCLS01] Jin Cao, William S. Cleveland, Dong Lin, and Don X. Sun. On the Nonstationarity of Internet Traffic. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'01)*, pages 102–112, 2001. 18

[CFSV99] Luigi Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. Performance Evaluation of the VF Graph Matching Algorithm. In *Proceedings of the 10th International Conference on Image Analysis and Processing (ICIAP'99)*, pages 1172–1177, 1999. 22

[CFSV01] Luigi Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. An Improved Algorithm for Matching Large Graphs. In *Proceedings of the 3rd Workshop on Graph-based Representations in Pattern Recognition*, pages 149–159, 2001. 22

[CFSV04] Luigi Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (Sub)graph Isomorphism Algorithm for Matching Large Graphs. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 26, pages 1367–1372, 2004. 22

[Che00] Shigeru Cheba. Load-time Structural Behavior in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00)*, pages 313–336. Springer, 2000. 49

[CIM+11] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. CloneCloud: Elastic Execution Between Mobile Devices and Cloud. In *Proceedings of the 6th European Conference on Computer Systems (Eurosys'11)*, pages 301–314. ACM, 2011. 12, 50

[CM09] Byung-Gon Chun and Petros Maniatis. Augmented Smartphone Applications Through Clone Cloud Execution. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS XII)*, pages 8–13, 2009. 12, 38

[CM10] Byung-Gon Chun and Petros Maniatis. Dynamically Partitioning Applications Between Weak Devices and Clouds. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing and Services: Social Networks and Beyond (MSC'10)*, pages 1–5. ACM, 2010. 39

[CPU09] SetCPU, 2009. http://www.setcpu.com/. 115

[CRB09] Mosharaf Kabir Chowdhury, Muntasir Raihan Rahman, and Raouf Boutaba. Virtual Network Embedding With Coordinated Node and Link Mapping. In *Proceedings of the 28th International Conference on Computer Communications (INFOCOM09)*, pages 783–791, 2009. 20

[CSLZ10] Gang Chen, Abdolhossein Sarrafzadeh, Chor Ping Low, and Liang Zhang. A Self-organization Mechanism Based on Cross-entropy Method

for P2P-like Applications. *ACM Transactions on Autonomous and Adaptive Systems*, 5:1–31, 2010. 22

[DB78] Peter J. Denning and Jeffrey P. Buzen. The Operational Analysis of Queuing Network Models. In *ACM Computing Surveys*, volume 10, pages 225–261, 1978. 15

[Dij74] Edgar W. Dijkstra. On the Role of Scientific Thought. In *Selected writings on computing: A personal perspective*, 1974. 23

[EC2] Amazon EC2. aws.amazon.com/ec2/. 39, 70, 137

[Ecl12] Eclipse, 2012. http://www.eclipse.org/. 56

[Emu11] Emulab Total Network Testbed, 2011. http://www.emulab.net/. 19

[EW97] Mary Mehrnoosh Eshaghian and Ying-Chieh Wu. Mapping Heterogeneous Task Graphs onto Heterogeneous System Graphs. In *Proceedings of the 6th Heterogeneous Computing Workshop (HCW'97)*, pages 147–160, 1997. 21

[FMK+10] Hossein Falaki, Ratul Mahajan, Srikanth Kandula, Dimitrios Lymberopoulos, Ramesh Govindan, and Deborah Estrin. Diversity in Smartphone Usage. In *Proceedings of the 8th International Conference on Mobile Systems, Applications and Services (MobiSys'10)*, pages 179–194. ACM, 2010. 66

[FPS02] Jason Flinn, So Young Park, and Mahadev Satyanarayanan. Balancing Performance, Energy and Quality in Pervasive Computing. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 217–226. IEEE, 2002. 10, 38

[FPV98] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998. 11

[Fra99] Greg Franks. Performance Analysis of Distributed Server Systems. *PhD Thesis, Carleton University*, 1999. 16

[Fra12] Fragments, 2012. android.com/guide/topics/fundamentals/fragments. Retrieved on August 2010. 39

[Fre10] FreeTTS, 2010. freetts.sourceforge.net/docs/index.php. Retrieved on October 2010. 45, 69

[GC04] Sachin Goyal and John Carter. A Lightweight Secure Cyber Foraging Infrastructure for Resource-constrained Devices. In *Proceedings of the 6th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'04)*, pages 186–195, 2004. 11

[GCTS12] Ioana Giurgiu, Claris Castillo, Asser Tantawi, and Malgorzata Steinder. Enabling Efficient Placement of Virtual Infrastructures in the Cloud. In *Proceedings of the ACM/IFIP/USENIX 13th International Conference on Middleware (Middleware'12)*, 2012. 122

[GD10] Richard Gass and Christophe Diot. An Experimental Performance Comparison of 3G and WiFi. In *Proceedings of the 11th International Conference on Passive and Active Measurement (PAM'10)*, pages 71–80, 2010. 51

[GHJ+09] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of the ACM SIGCOMM Conference on Data Communication (SIGCOMM'09)*, pages 51–62, 2009. 137

[Giu12] Ioana Giurgiu. Understanding Performance Modeling for Modular Mobile-Cloud Applications. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE'12)*, pages 259–262, 2012. 93

[GJP+06] Omprakash Gnawali, Ki-Young Jang, Jeongyeup Paek, Marcos Vieira, Ramesh Govindan, Ben Greenstein, August Joki, Deborah Estrin, and Eddie Kohler. The Tenet Architecture for Tiered Sensor Networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys'06)*, pages 153–166. ACM, 2006. 11

[GLL+09] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *Proceedings of the ACM SIGCOMM Conference on Data Communication (SIGCOMM'09)*, pages 63–74, 2009. 137

[GMP+06] Ben Greenstein, Christopher Mar, Alex Pesterev, Shahin Farshchi, Eddie Kohler, Judy Jack, and Deborah Estrin. Capturing High-frequency Phenomena using a Bandwidth-limited Sensor Network. In *Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys'06)*, pages 279–292. ACM, 2006. 11

[GMS05] Marco Gori, Marco Maggini, and Lorenzo Sarti. Exact and Approximate Graph Matching Using Random Walks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27:1100–1111, 2005. 22, 23

[GNM+03] Xiaohui Gu, Klara Nahrstedt, Alan Messer, Ira Greenberg, and Dejan Milojicic. Adaptive Offloading Inference for Delivering Applications in Pervasive Computing Environments. In *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications (PERCOM'03)*, pages 107–114. IEEE Computer Society, 2003. 11

[GRA12] Ioana Giurgiu, Oriana Riva, and Gustavo Alonso. Dynamic Software Deployment from Clouds to Mobile Devices. In *Proceedings of the ACM/IFIP/USENIX 13th International Conference on Middleware (Middleware'12)*. ACM, 2012. 39, 40, 95

[GRJ+09] Ioana Giurgiu, Oriana Riva, Dejan Juric, Ivan Krivulev, and Gustavo Alonso. Calling the Cloud: Enabling Mobile Phones as Interfaces to Cloud Applications. In *Proceedings of the ACM/IFIP/USENIX 10th International Conference on Middleware (Middleware'09)*, pages 1–20. Springer, 2009. 39, 40, 44, 95

[HBKK05] Kin-Ping Hui, Nigel Bean, Miro Kraetzl, and Dirk Kroese. The Cross-entropy Method for Network Reliability Estimation. *Annals of Operations Research*, 134:101–118, 2005. 22

[HF75] Griffith Jr. Hamlin and James D. Foley. Configurable Applications for Graphics Employing Satellites. In *Proceedings of the 2nd annual conference on Computer graphics and interactive techniques (SIGGRAPH '75)*, pages 9–19. ACM, 1975. 10

[HNR68] Peter Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions of Systems, Science and Cybernetics*, 4:100–107, 1968. 135

[HPC12] Amazon HPC, 2012. http://aws.amazon.com/hpc-applications/. 122

[HS99] Galen C. Hunt and Michael L. Scott. The Coign Automatic Distributed Partitioning System. In *Proceedings of the 3rd symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 187–200, 1999. 10

[HS03] Ningning Hu and Peter Steenkiste. Evaluation and Characterization of Available Bandwidth Probing Techniques. *IEEE Journal on Selected Areas in Communications*, 2003. 53

[HXT+10] Junxian Huang, Qjang Xu, Birdoth Tiwana, Zhuoqing Morley Mao, Ming Zhang, and Paramvir Bahl. Anatomizing Application Performance Differences on Smartphones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications and Services (MobiSys'10)*, pages 165–178. ACM, 2010. 49

[IOS12] Apple iOS, 2012. http://www.apple.com/ios/. 29

[Jai91] Raj Jain. The Art of Computer Systems Performance Analysis. In *John Wiley & Sons*, 1991. 15

[JdLT+95] Anthony D. Joseph, A. F. de Lespinasse, Joshua A. Tauber, David K. Gifford, and Frans Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP'95)*, pages 156–171, 1995. 12

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained Mobility in the Emerald System. 6(1):109–133, 1988. 11

[JPE+11] Deepal Jayasinghe, Calton Pu, Tamar Eilam, Malgorzata Steinder, Ian Whalley, and Ed Snible. Improving Performance and Availability of Services Hosted on IaaS Clouds With Structural Constraint-aware Virtual Machine Placement. In *Proceedings of the IEEE International Conference on Services Computing (SCC'11)*, pages 72–79, 2011. 21, 130

[KB03] Samuel Kounev and Alejandro Buchmann. Performance Modeling and Evaluation of Large-scale J2EE Applications. In *Proceedings of the International Conference on Computer Measurements (CMG'03)*, 2003. 16

[KBM+00] Tim Kindberg, John Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Philippe Debaty, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris, John Schettino, Bill Serra, and Mirjana Spasojevic. People, Places, Things: Web Presence for the Real World. In *Proceedings of the 3rd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'00)*, page 19, 2000. 11

[Kel05] Terence Kelly. Detecting Performance Anomalies in Global Applications. In *Proceedings of the 2nd Conference on Real Large Distributed Systems (WORLDS'05)*, pages 42–47, 2005. 108, 110

[Kes91] Srinivasan Keshav. A Control-theoretic Approach to Flow Control. In *Proceedings of ACM ACM Conference on Applications, Technologies, Architectures and Protocols for Computer Communication (SIGCOMM'91)*, pages 3–15. ACM, 1991. 53

[KK98] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998. 57

[KN01] Minkyong Kim and Brian Noble. Mobile Network Estimation. In *Proceedings of the 7th ACM Conference on on Mobile Computing and Networking (MobiCom'01)*. ACM, 2001. 53

[KRTP00] Nectarios Koziris, Michael Romesis, Panayiotis Tsanakas, and George Papakonstantinou. An Efficient Algorithm for the Physical Mapping of Clustered Task Graphs onto Multiprocessor Architectures. In *Proceedings of the 8th Euromicro Conference on Parallel and Distributed processing (EURO-PDP'00)*, pages 406–413, 2000. 20

[KT12] Young-Woo Kwon and Eli Tilevich. Power-efficient and Fault-tolerant Distributed Mobile Execution. In *Proc. of the 32nd International Conference on Distributed Computing Systems*, 2012. 14

[LB00] Kevin Lai and Mary Baker. Measuring Link Bandwidths Using a Deterministic Model of Packet Delay. In *Proceedings of ACM Conference on Applications, Technologies, Architectures and Protocols for Computer Communication (SIGCOMM'00)*, pages 283–294. ACM, 2000. 53

[LHS05] Xue Liu, Jin Heo, and Lui Sha. Modeling 3-tiered Web Applications. In *Proceedings of the 13th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)*, pages 307–310, 2005. 16

[LK09] Jens Lischka and Holger Karl. A Virtual Network Mapping Algorithm Based on Subgraph Isomorphism Detection. In *Proceedings of the 1st ACM Workshop on Virtualized Infrastructure Systems and Architectures (VISA'09)*, pages 81–88, 2009. 22

[LKL01] Te-Kai Liu, Santhosh Kumaran, and Zongwei Luo. Layered Queuing Models for Enterprise JavaBean Applications. In *Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing (EDOC'01)*, 2001. 16

[LLM11] Changbin Liu, Boon Thau Loo, and Yun Mao. Declarative Automated Cloud Resource Orchestration. In *Proceedings of the 2nd Symposium on Cloud Computing (SOCC'11)*, pages 1–8. ACM, 2011. 122

[LNP+03] Ron Levy, Jay Nagarajarao, Giovanni Pacifici, Mike Spreitzer, Asser Tantawi, and Alaa Youssef. Performance Management for Cluster Based Web Services. In *Proceedings of the IFIP/IEEE 8th International Symposium on Integrated Network Management*, pages 247–261, 2003. 16

[LT06] Jing Lu and Jonathan Turner. Efficient Mapping of Virtual Networks Onto a Shared Substrate. In *Technical Report WUCSE-2006-35, Washington University, USA*, 2006. 19

[Mae05] Maemo, 2005. www.maemo.org. Retrieved on December 2008. 29, 33, 39, 42

[MAFM99] Daniel A. Menasce, Virgilio F. Almeida, Rodrigo Fonseca, and Marco A. Mendes. A Methodology for Workload Characterization for E-commerce Sites. In *Proceedings of the 1st ACM Conference on Electronic Commerce (EC'99)*, pages 119–128, 1999. 17, 93

[MDA07] Gabor Madl, Mikil Dutt, and Sherif Abdelwahed. Performance Estimation of Distributed Real-time Embedded Systems by Discrete Events Simulations. In *Proceedings of the 7th ACM/IEEE International Conference on Embedded Software (EMSOFT'07)*, pages 183–192, 2007. 18

[Mey88] Bertrand Meyer. Object-oriented Software Construction. In *Prentice-Hall, Upper Saddle River, USA, 1st edition*, 1988. 24

[MGB+02] Alan Messer, Ira Greenberg, Phillipe Bernadat, Dejan Milojicic, Deqing Chen, T.J. Giuli, and Xiaohui Gu. Towards A Distributed Platform for Resource-constrained Devices. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 43–51, 2002. 11

[Mob11] MobiPerf, 2011. www.mobiperf.com/index.html. Retrieved on March 2011. 51

[MPF+09] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. PortLand: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric. In *Proceedings of the ACM SIGCOMM Conference on Data Communication (SIGCOMM'09)*, pages 39–50, 2009. 137

[MPZ10] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement. In *Proceedings of the 29th conference on Information Communications (INFOCOM'10)*, pages 1154–1162, 2010. 21, 122, 130

[MS97] Qingming Ma and Peter Steenkiste. On Path Selection for Traffic With Bandwidth Guarantees. In *Proceedings of IEEE International Conference on Network Protocols (ICNP'97)*, pages 191–202, 1997. 129

[MS10] Kathy Macropol and Ambuj Singh. Scalable Discovery of Best Clusters on Large Graphs. *Proceedings of the VLDB Endowment*, 3:693–702, 2010. 20

[NKNW96] John Neter, Michael Kutner, Christopher Nachtsheim, and William Wasserman. Applied Linear Statistical Models. In *Irwin, 4th edition*, 1996. 109

[NKSI05] Yang Ni, Ulrich Kremer, Adrian Stere, and Liviu Iftode. 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)*, pages 249–260. ACM, 2005. 11

[NSN+97] Brian Nobble, M. Satyanarayanan, D. Narayanan, James Eric Tilton, Jason Flinn, and Kevin Walker. Agile Application-aware Adaptation for Mobility. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP'97)*, pages 276–287. ACM, 1997. 12, 53

[NTG+09] Ryan Newton, Sivan Toledo, Lewis Girod, Hari Balakrishnan, and Samuel Madden. Wishbone: Prole-based Partitioning for Sensornet Applications. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI'09)*, pages 395–408, 2009. 11

[OI09] Eiji Oki and Ayako Iwaki. Performance Comparisons of Optimal routing by Pipe, Hose, and Intermediate Models. In *Proceedings of the 32nd International Conference on Sarnoff Symposium (SARNOFF'09)*, pages 104–108, 2009. 19

[OSG07] OSGi Alliance. *OSGi Service Platform, Core Specification Release 4, Version 4.1, Draft*, 2007. 25, 39, 42

[OYZ07] Shumao Ou, Kun Yang, and Jie Zhang. An Effective Offloading Middleware for Pervasive Services on Mobile Devices. *Pervasive Mobile Computing*, 3:362–385, 2007. 11

[Par72] David Lorge Parnas. On the Criteria to be Used in Decomposing Systems Into Modules. In *Communications of the ACM*, volume 15(12), 1972. 23

[Pat06] Pathload, 2006. www.cc.gatech.edu/fac/Constantinos.Dovrolis/bw-est/pathload.html. 53

[Pow95] Robert A. Powers. Batteries for Low Power Electronics. 83:687–693, 1995. 5

[Pow09] PowerTutor, 2009. www.ziyang.eecs.umich.edu/projects/powertutor/. Retrieved on May 2011. 53

[Pro11] ProSyst, 2011. www.prosyst.com/index.php/de/html/content/69/Mobile-Solutions-for-Mobile-Operators/. 42

[PS05] Joseph A. Paradiso and Thad Starner. Energy Scavenging for Mobile and Wireless Electronics. *IEEE Pervasive Computing*, 4(1):18–27, 2005. 2, 5

[PY10] Jing Tai Piao and Jun Yan. A Network-aware Virtual Machine Placement and Migration Approach in Cloud Computing. In *Proceedings of the 9th International Conference on Grid and Cloud Computing (GCC'10)*, pages 87–92, 2010. 21

[RA07] Jan S. Rellermeyer and Gustavo Alonso. Concierge: A Service Platform for Resource-Constrained Devices. In *Proceedings of the 2007 ACM EuroSys Conference (EuroSys'07)*, pages 245–258. ACM, 2007. 42

[RAL03] Robert Ricci, Chris Alfeld, and Jay Lepreau. A Solver for the Network Testbed Mapping Problem. *ACM SIGCOMM Computer Communication Review*, 33:65–81, 2003. 19

[RAR07] Jan S. Rellermeyer, Gustavo Alonso, and Timothy Roscoe. R-OSGi: Distributed Applications Through Software Modularization. In *Proceedings of the 8th International Conference on Middleware (Middleware'07)*, pages 1–20. Springer, 2007. 39, 42

[Rel11] Jan S. Rellermeyer. Modularity as a Systems Design Principle. *PhD Thesis, ETH Zurich*, 2011. 25, 27, 28, 171

[RK04] Reuven Rubinstein and Dirk P. Kroese. *The Cross-entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation and Machine Learning.* Information in Sciences and Statistics Series. Springer-Verlag New York, LLC, 2004. 22

[RKKD10] Jerry A. Rolia, Amir Kalbasi, Dikawar Krishnamurthy, and Stephen Dawson. Resource Demand Modeling for Multi-tier Services. In *Proceedings of the 1st Joint WOSP/SIPEW International Conference on Performance Engineering (WOSP/SIPEW'10)*, pages 207–216, 2010. 18

[RL80] Martin Reiser and Stephen S. Lavenberg. Mean-value Analysis of Closed Multichain Queuing Networks. *Journal of the Association for Computing Machinery*, 27:313–322, 1980. 98

[RR10] Adil Razzaq and Muhammad Siraj Rathore. An Approach Towards Resource Efficient Virtual Network Embedding. In *Proceedings of the 2nd International Conference on Evolving Internet (INTERNET 10)*, pages 68–73, 2010. 20

[RRA08] Jan S. Rellermeyer, Oriana Riva, and Gustavo Alonso. Alfredo: An architecture for flexible interaction with electronic devices. In *Proceedings of the 9th International Middleware Conference (Middleware'08)*, pages 22–41. Springer, 2008. 40

[RRB+03] Vinay J. Ribeiro, Rudolf H. Riedi, Richard G. Baraniuk, Jiri Navratil, and Les Cottrell. pathChirp: Efficient Available Bandwidth Estimation for Network Paths. In *Proceedings of Passive and Active Measurements Workshop (PAM'03)*, 2003. 53

[RS95] Jerry A. Rolia and Ken C. Sevcik. The Method of Layers. *IEEE Transactions in Software Engineering*, 21:689–700, 1995. 16

[RSDI05] Nishkam Ravi, Peter Stern, Niket Desai, and Liviu Iftode. Accessing Ubiquitous Services Using Smart Phones. In *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom'05)*, pages 383–393, 2005. 46

[RSF+06] Nishkam Ravi, Pravin Shankar, Andrew Frankel, Ahmed Elgammal, and Liviu Iftode. Indoor Localization Using Camera Phones. In *Proceedings of the 7th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'06)*, pages 1–7. IEEE Computer Society, 2006. 68

[RSM+11] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: Enabling Interactive Perception Applications on Mobile Devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications and Services*, 2011. 14

[RT99] C. Radhakrishna Rao and Helge Toutenburg. Linear Models: Least Squares and Alternatives. *Springer*, 1999. 110

[Rub99] Reuven Rubinstein. The Cross-entropy Method for Combinatorial and Continuous Optimization. *Methodology and Computing in Applied Probability*, 1:127–190, 1999. 22

[Sat01] Mahadev Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, 8(4):10–17, 2001. 10, 11

[SBCD09] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. The Case for VM-based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009. 14, 38

[SD76] Douglas C. Schmidt and Larry E. Druffel. A Fast Backtracking Algorithm to Test Directed Graphs for Isomorphism Using Distance Matrices. *Journal of the ACM*, 23:433–445, 1976. 22

[SD05] Soumya Sanyal and Sajal K. Das. MaTCH : Mapping Data-Parallel Tasks on a Heterogeneous Computing Platform Using the Cross-Entropy Heuristic. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPD'05)*, 2005. 22

[SF05] Ya-Yunn Su and Jason Flinn. Slingshot: Deploying Stateful Services in Wireless Hotspots. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications and Services (MobiSys'05)*, pages 79–92, 2005. 11

[SGBE05] Yunhe Shi, David Greee, Andrew Beatty, and Anton Ertl. Virtual Machine Showdown: Stack Versus Registers. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE'05)*, pages 153–163, 2005. 32

[SHD12] Choonsung Shin, Jin-Hyuk Hong, and Anind Dey. Understanding and Prediction of Mobile Application Usage for Smartphones. In *Proceedings of 14th ACM International Conference on Ubiquitous Computing (UbiComp'12)*, 2012. 4

[SIB03] Wayne Szeto, Youssef Iraqi, and Raouf Boutaba. A Multi-commodity Flow Based Approach to Virtual Network Resource Allocation. In *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM'03)*, volume 6, pages 3004–3008, 2003. 19

[Sil10] Microsoft Silverlight, 2010. www.silverlight.net. 39

[SJ99] Santhanam Srinivasan and Niraj K. Jha. Safety and Reliability Driven Task Allocation in Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(3):238–251, 1999. 21

[SKZ07] Christopher Stewart, Terence Kelly, and Alex Zhang. Exploiting Nonstationarity for Performance Prediction. In *Proceedings of the European Conference on Computer Systems (EUROSYS'07)*, pages 31–44, 2007. 15, 17, 18, 93, 107

[SMC74] Wayne Stevens, Glenfors Myers, and Larry Constantine. *Structured Design*, 13, 1974. 24

[SP09] Venu Satuluri and Srinivasan Parthasarathy. Scalable Graph Clustering Using Stochastic Flows: Applications to Community Discovery. In *Proceedings of the 15th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD'09)*, pages 737–746, 2009. 20

[Spe06] Speedtest, 2006. www.speedtest.net. Retrieved on March 2011. 51

[SPPM11] Simon Schwarzer, Patrick Peschlow, Lukas Pustina, and Peter Martini. Automatic Estimation of Performance Requirements for Software Tasks of Mobile Devices. In *Proceedings of the 2nd International Conference on Performance Engineering (ICPE'11)*, pages 347–358, 2011. 18, 93

[SRK07] Santonu Sarkar, Girish Maskeri Rama, and Avinash C. Kak. API-based and Information-theoretic Metrics for Measuring the Quality of Software Modularization. *IEEE Transactions on Software Engineering*, 33(1):14–32, 2007. 43

[SS05] Christopher Stewart and Kai Shen. Performance Modeling and System Management for Multi-component Online Services. In *Proceedings of the 2nd Conference on Networked Systems Design and Implementation (NSDI'05)*, pages 71–84, 2005. 17, 93, 108

[ST85] Chien-Chung Shen and Wen-Hsiang Tsai. A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion. *IEEE Transactions on Computers*, C-34(3):197–203, 1985. 22

[Sta75] George Merritt Stabler. *A System for Interconnected Processing*. PhD thesis, Providence, USA, 1975. 10

[Str05] Daniel W. Stroock. An Introduction to Markov Processes. *Springer*, 2005. 107

[SWHB06] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open Versus Closed: A Cautionary Tale. In *Proceedings of the 3rd Conference on Networked Systems Design and Implementation (NSDI'06)*, pages 18–31, 2006. 98

[SZdR+11] Raja R. Sambasivan, Alice X. Zheng, Michael de Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing Performance Changes by Comparing Request Flows. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, pages 4–4, 2011. 18

[SZL+11] Vivek Shrivastava, Petros Zerfos, Kang-Won Lee, Hani Jamjoom, Yew-Huey Liu, and Suman Banerjee. Application-aware Virtual Machine Migration in Data Centers. In *Proceedings of the 30th IEEE International Conference on Computer Communications (INFOCOM'11)*, pages 66–70, 2011. 21

[Tan12a] Asser Tantawi. A Scalable Algorithm for Placement of Virtual Clusters in Large Data Centers. In *Proceedings of the IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'12)*, 2012. 22

[TAN+12b] Narendran Thiagarajan, Gaurav Aggarwal, Angela Nicoara, Dan Boneh, and Jatinder Pal Singh. Who Killed My Battery: Analyzing Mobile Browser Energy Consumption. In *Proceedings of the 21st International Conference on World Wide Web (WWW'12)*, pages 41–50, 2012. 78

[TC00] Kenjiro Taura and Andrew A. Chien. A Heuristic Algorithm for Mapping Communicating Tasks on Heterogeneous Resources. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW'00)*, pages 102–115, 2000. 20, 130

[TDZN10] Eno Thereska, Bjoern Doebel, Alice X. Zheng, and Peter Nobel. Practical Performance Models for Complex, Popular Applications. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'10)*, pages 1–12, 2010. 18, 93

[TKSZ12] Chad Tossell, Philip Kortum, Ahmad Rahmatiand Clayton Shepard, and Lin Zhong. Characterizing Web Use on Smartphones. In *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems (CHI'12)*, pages 2769–2778, 2012. 4

[TPC05] TPC-W Benchmark, 2005. http://www.tpc.org/tpcw/. 107

[TSSP07] Chunqiang Tang, Malgorzata Steinder, Mike Spreitzer, and Giovanni Pacifici. A Scalable Application Placement Controller for Enterprise Data Centers. In *Proceedings of the 16th International Conference on World Wide Web (WWW'07)*, pages 331–340, 2007. 21

[Ull76] Julian R. Ullmann. An Algorithm for Subgraph Isomorphism. *Journal of the ACM*, 23:31–42, 1976. 22

[UPn00] UPnP. Universal Plug and Play Device Architecture. `http://www.upnp.org/`, 2000. 29

[UPS+05] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An Analytical Model for Multi-tier Internet Services and Its Applications. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'05)*, pages 291–302, 2005. 15, 16, 17, 93

[UPS+07] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. Analytic Modeling of Multitier Internet Applications. *ACM Transactions on the Web*, 1, 2007. 15, 16, 17, 93

[USR02] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*, pages 239–254, 2002. 16

[Van10] Bob Vandette. Java SE Embedded Performance Versus Android 2.2. *Oracle Corporation*, 2010. 32

[vD00] Stijn van Dongen. Graph Clustering by Flow Simulation. *PhD Thesis, University of Utrecht, Utrecht*, 2000. 20

[VTM09] Hien Nguyen Van, Frederic Dang Tran, and Jean-Marc Menaud. Autonomic Virtual Resource Management for Service Hosting Platforms. In *Proceedings of the ICSE Workshop on Software Engineering Challenges of Cloud Computing (ICSE'09)*, pages 1–8, 2009. 21

[Wei91] Mark Weiser. The Computer for the Twenty-First Century. *Scientific American*, 265(3):94–104, September 1991. 11

[Wil05] Rand Wilcox. Introduction to Robust Estimations and Hypothesis Testing. *Elsevier, 2nd edition*, 2005. 110

[Win12] Windows Mobile, 2012. http://www.microsoft.com/windowsphone/en-us/default.aspx. 29

[WMG+10] Brian J. Watson, Manish Marwah, Daniel Gmach, Yuan Chena dn Martin Arlitt, and Zhikui Wang. Probabilistic Performance Modeling of Virtualized Resource Allocation. In *Proceedings of the 7th International Conference on Autonomic Computing (ICAC'10)*, pages 99–108, 2010. 18, 93

[WPD+02] Roy Want, Trevor Pering, Gunner Danneels, Muthu Kumar, Murali Sundar, and John Light. The Personal Server: Changing the Way We Think about Ubiquitous Computing. In *Proceedings of the 4th international conference on Ubiquitous Computing (UbiComp'02)*, pages 194–209. Springer-Verlag, 2002. 11

[WR95] Murray Woodside and Govindachari Raghunath. General Bypass Architecture for High-performance Distributed Algorithms. In *Proceedings of the 6th IFIP Conference on Performance of Computer Networks*, pages 51–65, 1995. 16

[XOWM06] Jing Xu, Alexandre Oufimtsev, Murray Woodside, and Liam Murphy. Performance Modeling and Prediction of Enterprise Java Beans with Layered Queuing Network Templates. *ACM SIGSOFT Software Engineering Notes*, 31, 2006. 16

[YCL95] Ching-Wei Yeh, Chung-Kuan Cheng, and Ting-Ting Y. Lin. Circuit Clustering Using a Stochastic Flow Injection Method. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(2):154–162, 1995. 20

[YGG+01] Fan Yang, Nitin Gupta, Nicholas Gerner, Xin Qi, Alan Demers, Johannes Gehrke, and Jayavel Shanmugasundaram. Computation Offloading to Save Energy on Handheld Devices: A Partition Scheme. In *Proceedings of the 2001 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'01)*, pages 238–246. ACM, 2001. 12

[YGG+07] Fan Yang, Nitin Gupta, Nicholas Gerner, Xin Qi, Alan Demers, Johannes Gehrke, and Jayavel Shanmugasundaram. A Unified Platform for Data Driven Web Applications With Automatic Client-server Partitioning. In *Proceedings of the 16th International World Wide Web Conference (WWW'07)*, pages 341–350. ACM, 2007. 12

[YL01] Cliff Young and Yagati N. Lakshman. Protium, An Infrastructure for Partitioned Applications. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS VIII)*, pages 47–52, 2001. 12

[YL03] Cliff Young and Yagati N. Lakshman. Bandwidth Estimation: Metrics, Measurements, Techniques and Tools. *IEEE Networks*, 17(6):27–35, 2003. 53

[YR12] Qin Yin and Timothy Roscoe. VF2x: Fast, Efficient Virtual Network Mapping for Real Testbed Workloads. In *Proceedings of the 8th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TRIDENTCOM'12)*, 2012. 22

[YSRG06] Fan Yang, Jayavel Shanmugasundaram, Mirek Riedewald, and Johannes Gehrke. Hilda: A High-level Language for Data-driven Web Applications. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, pages 32–43. IEEE Computer Society, 2006. 12

[YYRC08] Minlan Yu, Yung Yi, Jennifer Rexford, and Mung Chiang. Rethinking Virtual Network Embedding: Substrate Support for Path Splitting and Migration. *ACM SIGCOMM Computer Communication Review*, 38:17–29, 2008. 19

[ZA06] Yong Zhu and Mostafa Ammar. Algorithms for Assigning Substrate Network Resources to Virtual Network Components. In *Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM'06)*, pages 1–12, 2006. 19, 127, 128, 137, 144, 173

[ZJKG10] Xinwen Zhang, Sangoh Jeong, Anugeetha Kunjithapatham, and Simon Gibbs. Towards An Elastic Application Model for Augmenting Computing Capabilities of Mobile Platforms. In *Proceedings of the 3rd International ICST Conference on Mobile Wireless Middleware, Operating Systems and Applications (Mobilware'10)*, pages 270–284, 2010. 14

[ZSB+08] Xiaoyun Zhu, Cipriano Santos, Dirk Beyer, Julie Ward, and Sharad Singhal. Automated Application Component Placement in Data Centers Using Mathematical Programming. *Journal on Network Management*, 18:467–483, 2008. 19

[ZSG+09] Xinwen Zhang, Joshua Schiffman, Simon Gibbs, Anugeetha Kunjithapatham, and Sangoh Jeong. Securing Elastic Applications on Mobile Devices for Cloud Computing. In *Proceedings of the ACM Workshop on Cloud Computing Security (CCSW'09)*, pages 127–134, 2009. 14

[ZTQ+10] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *Proceedings of the 8th International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS'10)*, pages 105–114, 2010. 53

# List of Figures


1.1 Examples of applications: interior decoration, augmented reality, panorama generator and ticket machine (left to right, top to bottom). 3

2.1 OSGi architectural overview. . . . . 26
2.2 R-OSGi architectural overview (adapted from (Rel11)). . . . . 28
2.3 Android architectural overview. . . . . 30
2.4 Maemo architectural overview. . . . . 34

3.1 AlfredO architecture and pipeline of operators. . . . . 41
3.2 Data transfer and energy consumption comparison between *AlfredO bootstrap*, *AlfredO bootstrap + interactions* and *Full code pre-installation* for 10000 clients, during 1, 6 and 12 months (WiFi and 3G are used). . . . . 46
3.3 Example of bundle deployment using R-OSGi. . . . . 47
3.4 Example of application consumption graph. . . . . 56

4.1 Screenshots of prototype applications on HTC Desire (a–c) and on Nokia N810 (d). . . . . 66
4.2 Modularization of the interior decoration application. . . . . 67
4.3 Modularization of the ticket machine application. . . . . 68
4.4 Modularization of the indoor localization application. . . . . 69
4.5 Modularization of the text-to-speech synthesizer application. . . . 70
4.6 Startup time and power consumption on HTC Desire for the ticket machine, indoor localization and text-to-speech applications on EC2 US-East instances (WiFi and 3G used). . . . . 72
4.7 Overall interaction time (a) and resource consumption (b) for the interior decoration application (WiFi and bluetooth used). . . . . 74
4.8 Interaction time for indoor localization application with varying configurations, on different EC2 instances (WiFi and 3G used). . . 75
4.9 Interaction time for the ticket machine application with varying configurations, on different EC2 instances (WiFi and 3G used). . . 76
4.10 Interaction time for the text-to-speech application with varying configurations, on different EC2 instances (WiFi and 3G used). . . 77

4.11 Power consumption for the ticket machine, indoor localization and text-to-speech applications with varying configurations, on EC2 US-Large instances (WiFi and 3G used). . . . . 78
4.12 Overall time with multiple service invocations with WiFi. . . . . 80
4.13 Parallel bundle initialization for the localization application with 3 consecutive user interactions (3G in use). . . . . 82
4.14 Reactivity to changes in CPU load for the localization and text-to-speech applications (WiFi is used). . . . . 83
4.15 Reactivity to changes in network bandwidth for the indoor localization application (WiFi and 3G used). . . . . 84
4.16 Reactivity to changing user inputs for the indoor localization and text-to-speech applications (WiFi and 3G are used). . . . . 85
4.17 Opportunistic vs. baseline profiling in a 10-minute scenario with the indoor localization application (WiFi used). . . . . 86
4.18 Processing time for `ALL`, `1-Step`, `3-Step` and `5-Step`. . . . . 88

5.1 Comparison between the measured optimal distribution and the configuration chosen by AlfredO in various execution scenarios of the indoor localization application when extensive statistics are not available (WiFi used). . . . . 92
5.2 Integration of the performance estimation model with AlfredO. . . 94
5.3 Modeling an example modular application with queuing networks. 95
5.4 WiFi (a–b) and 3G (c–d) measured and estimated transfer times for Amazon EC2 small and large instances. . . . . 100
5.5 The modularization scheme for the image processing application. . 110
5.6 Comparison between estimated interaction times and collected measurements for all 27 valid distribution schemes for the image processing application. . . . . 111
5.7 Comparison between estimated interaction times and collected measurements for all 27 valid distribution schemes for the image processing application on a large EC2 instance. . . . . 112
5.8 Comparison between estimated interaction times and collected measurements when the CPU load on HTC Desire is raised to 50–60%. 113
5.9 Comparison between estimated interaction times and collected measurements when the CPU load on HTC Desire is raised to 50–60% and a large instance is used. . . . . 114
5.10 Comparison between estimated interaction times and collected measurements when the HTC Desire is replaced with a Motorola Droid and its CPU load is increased at 50–60%. . . . . 115
5.11 CDF percentage error per time interval $t$ for both WiFi and 3G. . 118

6.1 Placement process of virtual networked applications. . . . . 126
6.2 Cold spot discovery and VNI clustering examples. . . . . 130

6.3 Cold spot selection example for clustered VNI and a data center with 16 PMs. . . . . 133
6.4 Results for the generic VNI workload with various data center sizes. 138
6.5 Network performance with vs. without VNI clustering (a) and cold spot vs. data center level (b-d) in a 256 PMs data center. . . . . 139
6.6 Random cold spot selection vs. our default algorithm in a 256 PMs data center. . . . . 140
6.7 Comparison between random cold spots and our default algorithm on 256 PMs data center. . . . . 141
6.8 Topologies for the cache, hadoop and three-tiered VNIs. . . . . 141
6.9 Results for mix of cache, hadoop and three-tiered VNIs with various data center sizes. . . . . 142
6.10 Results with different threshold values in a 1024 PMs data center. 143
6.11 Comparison between our approach and the VNM technique proposed in (ZA06) for a 256 PMs data center. . . . . 144

## LIST OF FIGURES

# List of Tables


2.1 Main features summary for MAUI, CloneCloud and our approach. 13

3.1 WiFi and 3G download speeds, upload speeds and latencies experienced at different times during 12 hours (average and [standard deviation]). . . . . 51
3.2 WiFi and 3G download and upload throughput experienced while transferring 10KB, 50KB, 100KB, 200KB, 500KB, 1MB, 2MB and 3MB worth of data. . . . . 52

4.1 Startup time (average and [standard deviation]) with bluetooth for selected configurations of the interior decoration application. . . . 71
4.2 AlfredO performance and power consumption with the ticket machine, indoor localization and text-to-speech applications on large EC2 instances. . . . . 79
4.3 AlfredO's improvement over battery drop with the ticket machine, indoor localization and text-to-speech applications on large EC2 instances. . . . . 80
4.4 `ALL` and `K-Step` performance for single and multiple interactions with the interior decoration application. . . . . 87

5.1 CPU and network resource capacities comparison between (Motorola Droid, Amazon EC2 large) and (HTC Desire, Amazon EC2 small) setups. . . . . 102
5.2 Comparison between AlfredO ideal, the model and AlfredO default. 116
5.3 Summary of data set used for calibration and test phases. . . . . 117
5.4 Retrospective and prospective accuracy on the first half and second half of the data set, respectively. . . . . 118

6.1 Common terminology used in the placement problem. . . . . 124

# Curriculum Vitae

## EXPERIENCE

9.2008 – present **ETH Zurich, Switzerland**, *Research Assistant.*
**PhD Program**

- Designed and developed advanced systems for modern mobile platforms
- Designed and implemented a resource allocation system for cloud data centers in collaboration with IBM Watson Research Center, NY, USA
- Teaching assistant
- Supervized students writing Bachelor and MSc theses
- Published research results
- Attended conferences and summer schools

7.2011 – 10.2011 **IBM Watson Research Center, NY, USA**, *Research Intern.*
Optimization algorithms for resource allocation in large data centers)

- Worked with the management layer of the virtualization group
- Provided support for integration with IBM RC2 cloud

3.2008 – 8.2008 **DERI, Galway, Ireland**, *Research Intern.*
7.2007 – 9.2007 Completed design for semantic e-mail project (Semanta)

- Developed semantic e-mail for Mozilla Thunderbird e-mail client
- Integrated semantic e-mail in the EU Nepomuk project
- Served as instructor for building Thunderbird plugins

## EDUCATION

9.2008 – 9.2012 **PhD in Computer Science**, *ETH*, Zurich.
PhD thesis: "Integrating Cloud Applications with Mobile Devices: Design Principles and Performance Optimizations"

10.2003 – 6.2008 **Engineering Diploma**, *Technical University*, Cluj-Napoca, Romania.
Grade 9.84/10.00, Top of the class

## PUBLICATIONS

Middleware'12 *Dynamic Software Deployment from Clouds to Mobile Devices.* Ioana Giurgiu, Claris Castillo, Asser Tantawi and Malgorzata Steinder

Middleware'12 *Dynamic Software Deployment from Clouds to Mobile Devices.* Ioana Giurgiu, Oriana Riva and Gustavo Alonso

ICPE'12 *Understanding performance modeling for modular mobile-cloud applications.* Ioana Giurgiu

Eurosys'10 *Mobile phones as a gateway to cloud computing.* Ioana Giurgiu Doctoral Symposium

Middleware'09 *Calling the cloud: Enabling mabile phones as interfaces to cloud applications.* Ioana Giurgiu, Oriana Riva, Dejan Juric, Ivan Krivulev and Gustavo Alonso.

## PROJECTS

AlfredO *Dynamic Software Deployment from Clouds to Mobile Devices*

Developed a framework for dynamic and flexible deployments of distributed applications from clouds to Android and Maemo platforms, to improve performance and battery life on mobile devices

Aprilia *Efficient resource allocation for virtualized and networked applications in large data centers*

Developed a system to efficiently manage the placement of complex applications in large data centers which exploits graph topologies to consider performance and robustness constraints

## TEACHING EXPERIENCE

Spring 2009 – 2012 Advanced Computer Networks

Fall 2011 Advanced Systems Lab

## LANGUAGES

English Proficient (CAE, 2005)

French Intermediate level (level B1)

German Basic knowledge (level A1)

Spanish Intermediate level (level B1)

Romanian Mother tongue

## PERSONAL DEVELOPMENT

2011 **Finalist, Anita Borg Scholarship for Women in Computer Science**, *Google*.
Europe, the Middle East and Africa Universities

2001 – 2003 **Participation to Mathematics contests**, *Romania*.
Won 3rd and 4th places in all years