Adaptive Application Configuration and Distribution in Mobile Cloudlet Middleware

Tim Verbelen[1], Pieter Simoens[1,2], Filip De Turck[1], and Bart Dhoedt[1]

[1] Ghent University - IBBT, Department of Information Technology
[2] Ghent University College, Department INWE

Abstract. Despite recent advances in mobile device capabilities in terms of CPU power, memory, connectivity, etc, these devices still fall short to execute complex media rich and data analysis applications. Therefore, the concept of cloudlets was introduced, where nearby infrastructure is used by the mobile user for code offloading. However, the way this infrastructure is used is often left to the application developer, leading to a best effort approach in utilizing remote resources. In this paper we present a middleware approach for such cloudlet environments, that manages mobile applications on a component level. The middleware monitors application components in the cloudlet, and optimizes both the configuration and the deployment of all components in the cloudlet for the current execution context. We present a prototype implementation of the middleware platform, and show the effectiveness of our adaptation strategy using an augmented reality use case.

1 Introduction

Nowadays, mobile computing devices are becoming widespread given the increasing popularity of smartphones. Gartner reports that although worldwide sales of mobile phones declined by 2% during the first quarter of 2012, smartphone sales increased by 44.7% [4]. People no longer only use their mobile device for telephony, but also for a myriad of other mobile applications offered, such as location based services, multimedia applications, games and many more.

Despite many advances in technology, mobile devices will always be resource poor, as restrictions on weight, size, battery life, and heat dissipation impose limitations on computational resources and make mobile devices more resource constrained than their non-mobile counterparts [13]. Therefore, mobile devices still fall short to execute many media rich and data analysis applications that require heavy computation, and often also have (near) real-time constraints such as augmented reality (AR).

To address the resource limitations of mobile devices, cloud computing can be leveraged to offload tasks to the infrastructure of public cloud providers [5]. However, Hassan et al. [7] show that cloud computing is not a silver bullet, and is outperformed by outsourcing to nearby residential computers. Depending on the use case, outsourcing to the cloud can even be slower than local execution on the mobile device due to limited bandwidth and high WAN latencies. Therefore,

C. Borcea et al. (Eds.): MobilWare 2012, LNICST 65, pp. 178–191, 2013.
© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2013

Satyanarayanan [13] introduced the concept of VM based cloudlets: trusted, resource rich computers in the near vicinity of the mobile user (e.g. near or co-located with the wireless access point), on which virtual machines (VMs) are instantiated for remote execution.

Instead of adopting virtual machines as the unit of deployment, we choose a more fine grained approach where applications are managed on a component level [16]. This approach offers a number of advantages. First, the component management middleware allows for a more fine grained optimization than an "all or nothing" approach using VMs. Second, starting and migrating a component is an order of magnitude faster than starting and provisioning a virtual machine. Third, resources are managed by the middleware, which allows for dynamic discovery of resources in the network, that can join or leave the cloudlet at runtime. Finally, the middleware can optimize the component distribution and configuration for all users involved in the cloudlet, and optimally coordinate the allocation of resources that should be shared by multiple end users.

Adopting a fine-grained, component level approach however poses a number of issues. In addition to deciding on where to deploy, components should also be configured to run optimally on the available resources. This typically involves setting configuration parameters of components, such that the application is perceived to run at good quality. To achieve this, components can be specified to gracefully degrade when executed on low-end hardware and to perform better when they can exploit additional resources. When aiming for optimal application quality, constraints concerning total CPU- and network load should be satisfied, as well as timing constraints defined by the application developer. The problem at hand is therefore to solve the deployment and configuration problem, subject to both infrastructure and application constraints.

In this paper we present a component based middleware architecture, that configures and distributes application components at runtime. We propose a model driven middleware decision algorithm that optimizes both the application configuration and distribution, taking into account the network connectivity, the available resources and application constraints imposed by the application developer. To show the effectiveness of our approach, we use a mobile augmented reality application.

The remainder of this paper is structured as follows. In the next section, we discuss related work in the domain of code offloading. Section 3 describes in detail our cloudlet middleware architecture. In Section 4 a mathematical model is presented for the infrastructure, the application behavior and the application constraints. A heuristic algorithm is proposed to search for the global optimum. The algorithm is then evaluated in Section 5 using a mobile augmented reality application. Finally we conclude this paper in Section 6 and discuss future work.

2 Related Work

Offloading computation from mobile devices to remote resources has been a research topic for over a decade [1]. Several systems exist, offloading either at class, method, component or virtual machine level.

Ou et al. [12] present an adaptive offloading framework for offloading Java classes, in combination with a (k+1) partitioning algorithm. The fine granularity of class offloading however requires extensive monitoring and causes significant overhead.

Other systems use methods as units to outsource, such as the Scavenger cyber foraging system [10], which outsources Python methods. A dual-profile scheduler is used, weighting tasks according to their parameter input sizes and run time. MAUI [3] outsources method calls on the Microsoft .Net runtime environment. This platform generates a program partitioning by formulating and solving an integer linear programming problem to maximize energy savings.

A more coarse grained approach is to outsource software components. Zhang et al. [17] offloads platform independent software components – called weblets – to the cloud using a Bayesian learning scheduler. Giurgiu et al. [5] and Verbelen et al. [15] use OSGi components as units to outsource. To distribute these components, a graph model of the software is built and graph cutting algorithms are used to calculate the most appropriate deployment.

Goyal et al. [6] propose the use of virtualization on the infrastructure for remote execution. Here a client can request a virtual machine (VM) with specific resource guarantees to offload services to. Su et al. present Slingshot [14], where the VMs are co-located with the wireless access point to overcome the WAN latency. Chun et al. present CloneCloud [2], where virtualized clones of the mobile device are executed in the cloud. Different binaries of the application are generated in an off-line profiling stage, with special VM instructions added at migration points for selected methods. At runtime a clone VM is instantiated at the server side, and the application transparently switches between execution at the device or at the clone.

Satyanarayanan et al. [13] propose the concept a of cloudlet: a trusted, resource-rich computer or a cluster of computers well connected to the Internet and available for use by nearby mobile devices. Cloudlets offer their resources to mobile devices by dynamic VM synthesis, where small VM overlays are sent to the cloudlet from which a complete VM is created.

All these systems aim to optimize application execution solely by offloading. In this paper we combine the offloading problem with dynamic configuration adaptation, which allows the application to gracefully degrade when no or insufficient remote resources are available. All these systems also tackle the case of one mobile device offloading to one or more remote devices. In this contribution, we state a general optimization problem that also takes into account multiple mobile users sharing the same network and CPU resources.

3 Cloudlet Middleware

We envision the cloudlet architecture as shown in Figure 1, with three layers: the component level, the node level and the cloudlet level.

A component is the unit of deployment and is specified by its providing and required interfaces. Components are managed by an *Execution Environment (EE)*,

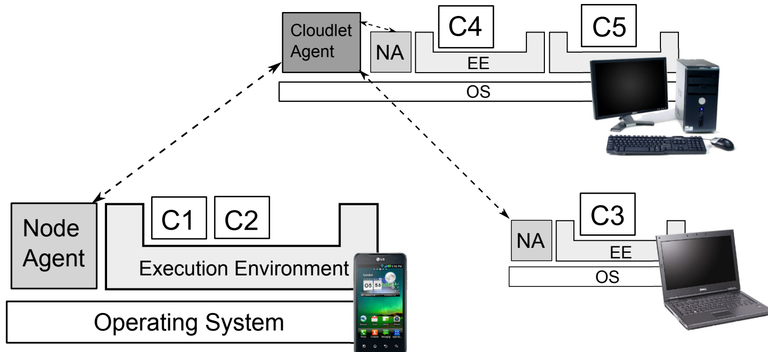


Fig. 1. The application components are distributed among nodes in the cloudlet, consisting of a mobile phone, a laptop and a desktop computer. All components are managed and monitored by an Execution Environment (EE). Different EEs on a node are managed by a Node Agent (NA), that in turn communicate with the Cloudlet Agent (CA).

that can start and stop components, resolve component dependencies, expose provided interfaces etc. To support distributed execution, dependencies can be resolved with other (remote) Execution Environments. In that case, proxies and stubs are generated and the components can communicate by remote procedure calls (RPCs). Components can also define performance constraints (e.g. the maximum execution time of a method), and expose configuration parameters to the EE. By monitoring the resource usage of each component, the EE can assess the behavior and the performance of the application, and detect violations of the imposed performance constraints.

Multiple EEs can run on top of an operating system (OS), which in turn can run on both virtualized or real hardware. The (possibly virtualized) hardware together with the installed OS is called a node, and is managed by a *Node Agent (NA)*. The Node Agent manages all the EEs running on the OS, and can also start or stop new Execution Environments, for example for sandboxing components. The NA also monitors the resource usage of the node as a whole, and has info about the (maybe virtualized) hardware it runs on (e.g. the number of processing cores, processing speed, etc.).

Multiple nodes that are in the physical proximity of each other (i.e. low latency) form a cloudlet. The cloudlet is managed by a *Cloudlet Agent (CA)*, that communicates with all underlying Node Agents. Nodes can dynamically join or leave the cloudlet, and are discovered using a service discovery protocol. Within one cloudlet, the node with the most resources is chosen to host the Cloudlet Agent.

The Cloudlet Agent has a global overview of all application components running on the different EEs, and contains the decision algorithm to optimize the deployment and configuration of all components in the cloudlet. This decision algorithm is triggered when an event occurs in the cloudlet, e.g. when a new device joins the cloudlet, when an EE detects a constraint violation, etc.

4 Decision Algorithm

We first present mathematical application and infrastructure models that capture all monitor information and are used to define constraints and an objective function to optimize. Because the solution space is too large to calculate the absolute optimum in a timely manner, we also present a heuristic to calculate a local optimum fast.

4.1 Application Model

An application consists of a number of components, that can offer a number of methods as service interface. An example application consisting of five components is shown in Figure 2. The arrows denote call dependencies, for example component C1 calls method m1 from component C2, which on its turn calls method m3 and method m4 of component C3. However, to take a decision on how to deploy the components, more information is needed on the actual control flow of the application.

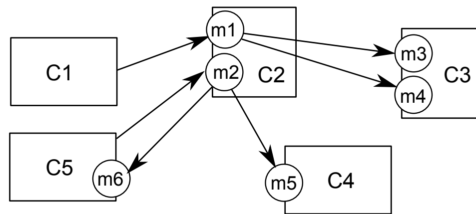


Fig. 2. An example component based application. Each component offers a number of methods in a service interface. Components communicate with each other by calling these service methods.

To capture the actual control flow of the application, we use sequence diagrams for all the scenarios of the application. For example, the sequence diagrams of the application presented in Figure 2 are shown on Figure 3.

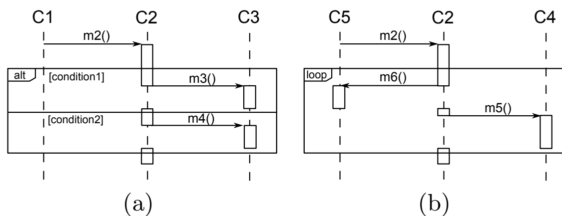


Fig. 3. The actual behavior of the application is captured in UML sequence diagrams

However, the sequence diagrams depicted in Figure 3 still fall short to describe the application behavior in sufficient detail. For example, in Figure 3(a) the total execution time before the call of method m2 by component C5 returns, depends on the number of times the loop is executed, and in Figure 3(b) the execution depends on the conditional path taken.

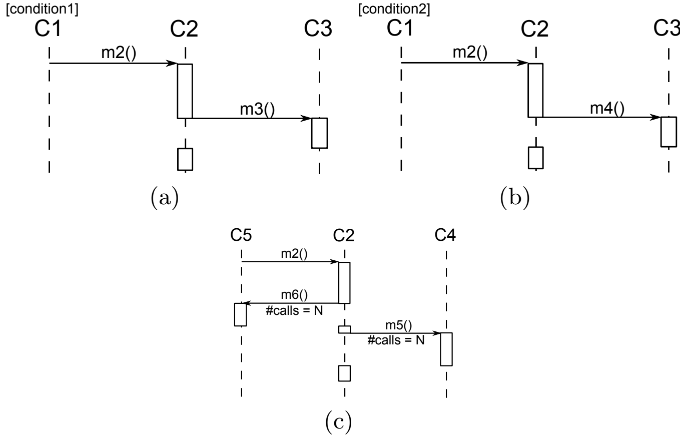


Fig. 4. The two UML sequence diagrams shown in Figure 3 are split up in 3 sequences. The loop is replaced by an annotation how many times each method is called within the sequence, and conditional sequences are split up in a separate sequence for each condition.

Therefore, sequences are represented as shown in Figure 4. To model the loop, the method calls in a sequence are annotated with the number of times they are called within the sequence as shown in Figure 4(a). The conditional sequence in Figure 3(b) is split up in multiple sequences (Fig. 4(b) and Fig. 4(c)), each representing one conditional path. To capture the overall application behavior, we also keep track of the number of times each sequence is called per time unit.

More formally, let C and M represent the set of application components and the set of public methods offered by all components. A sequence $s \in S(C, M)$ represents a sequence of calls of methods $m \in M$ between application components $c_i, c_j \in C$. $m_{sc_ic_j}$ denotes a call to method m of component c_j in sequence s by component c_i. To further define the application behavior $\#calls_s$ is the number of times sequence s is executed per time unit, and $\#calls_{m_{sc_ic_j}}$ is the number of times method call $m_{sc_ic_j}$ is executed in sequence s.

Finally, for each call $m_{sc_ic_j}$ we also track the size of the arguments of the method $A_{m_{sc_ic_j}}$, as well as the size of the return value $R_{m_{sc_ic_j}}$ and the relative CPU load $Load_{m_{sc_ic_j}}$ of the method call. The argument size, return size and CPU load of a method call $m_{sc_ic_j}$ are to be expressed as a function of the configuration parameters, which can be given by the developer, or can be estimated from monitoring information.

4.2 Infrastructure Model

The cloudlet consists of a number of interconnected devices $d \in D$. Each device processor has a rate at which load can be processed $CPUspeed_d$ and a number of cores $\#CPUcores_d$.

The devices are connected by a (wireless) network, that is characterized by its bandwidth BW and latency Lat. The bandwidth denotes both the capacity (maximum number of bytes that can be sent per time unit) as the speed (the rate at which bytes are sent) of the network. The latency is the round trip delay of the network.

4.3 Constraints

A number of constraints are defined that restrict the number of allowed deployments and configurations. The network is limited in capacity by the maximum number of bytes that can be sent per time unit, and also the devices have a maximum load that can be processed per time unit. In addition to the constraints imposed by the infrastructure capabilities, the application developer can also define constraints on the execution time of methods, for example restricting the maximum execution time of a method.

Let X_{id} be defined as

$$X_{id} = \begin{cases} 1 & \text{if component } c_i \text{ is deployed on device } d \\ 0 & \text{otherwise} \end{cases}$$

and $h_{ij} = 1 - \sum_d X_{id} \times X_{jd}$, meaning that h_{ij} equals 1 when c_i and c_j are deployed on a different device.

The bandwidth used (the number of bytes sent over the network per time unit) should be less than BW or

$$bandwidth = \sum_s \sum_m \sum_i \sum_j h_{ij} \times (A_{m_{sc_ic_j}} + R_{m_{sc_ic_j}}) \times \#calls_{m_{sc_ic_j}} \times \#calls_s \\ \leq BW$$

We assume that all methods called in the same sequence run on the same thread, and thus the load generated by a sequence on one device $load_{sd}$ should not exceed the maximum load that can be processed per time unit by one core or thus $\forall d$:

$$load_{sd} = \sum_m \sum_i \sum_j X_{jd} \times Load_{m_{sc_ic_j}} \times \#calls_{m_{sc_ic_j}} \times \#calls_s \\ \leq CPUspeed_d$$

Also, for each device the maximum load should not exceed the maximum load that can be processed per time unit on the whole device or $\forall d$:

$$\begin{aligned} load_d &= \sum_s load_{sd} \\ &\leq CPUspeed_d \times \#CPUcores_d \end{aligned}$$

Note that this is only an approximation of the maximum load of the device, as this also depends on the internal thread scheduling. However, we employ this constraint for simplicity, and because this already gives sufficient results.

Finally for each constrained method m the execution time of a method call $T_{m_{sc_ic_j}}$ should be lower than the imposed threshold or $\forall s, c_i$:

$$\begin{aligned} T_{m_{sc_ic_j}} = &(\sum_d X_{jd} \times Load_{m_{sc_ic_j}} \times \frac{1}{CPUspeed_d}) \\ &+h_{ij} \times ((A_{m_{sc_ic_j}} + R_{m_{sc_ic_j}}) \times \frac{1}{BW} + Lat) \\ &+ \sum_{m \in children(m_{sc_ic_j})} T_{m_c} \\ \leq\ & threshold_m \end{aligned}$$

4.4 Optimization Objective

The optimization objective is to maximize the utility of all components, where the utility function denotes the quality of the end user as a function of the configuration parameters:

$$max \sum_j utility_{c_j}(config\,params)$$

This utility function can be provided by the application developer. In this paper, we use the load generated by all methods of the component as utility measure, assuming that more work done by the component results in a better quality or $\forall c_j$:

$$utility_{c_j}(config\,params) = \sum_s \sum_m \sum_i Load_{m_{sc_ic_j}}$$

However, also another utility function could be used, for example one could define an utility function for minimizing the energy usage, when the devices energy characteristics are known (i.e. energy usage per CPU load, energy usage per byte received/sent, etc.).

4.5 Optimization Algorithm

To find the optimal configuration and deployment, the goal is to find an assignment of each component to a device, and a value for each configuration parameter that optimizes the utility function, while adhering to all imposed constraints. In the situation of d devices, c components, p parameters and v_p possible values for parameter p, the number of possible solutions is $d^c \times \prod_p v_p$. Therefore, a brute force search for the optimum is inappropriate for use at runtime due to the long calculation time. To find a valid (although possibly suboptimal) solution in acceptable time, we use the heuristic explained in pseudocode in algorithm 1.

The algorithm is inspired by the KL graph partitioning algorithm [8], and consists of two loops. The outer loop continues until no better solution is found. The inner loop calculates a number of possible "moves" in solution space. A possible move is an increase or decrease of a configuration parameter value, or a migration of a component to another device. For all possible moves, an objective function is evaluated, and the gain is calculated as the difference with the objective of the current best solution. Subsequently, the move with the highest gain is performed and a new solution is found. The performed move is kept in an *ExploredMoves* list, that ensures that this move is not repeated later on in the loop.

Algorithm 1. Configuration and deployment decision algorithm

```
CurrentSolution ← StartSolution
BestSolution ← StartSolution
repeat
  ExploredMoves ← InitialMoves
  repeat
    Calculate possible moves K such that ∀k ∈ K : k ∉ ExploredMoves
    Calculate objective gain g, ∀k ∈ K
    Perform move k_best with maximum gain g to get NewSolution
    CurrentSolution ← NewSolution
    Add k_best to ExploredMoves
    if objective(BestSolution) < objective(CurrentSolution) then
      BestSolution ← CurrentSolution
    end if
  until no more moves possible
until no better solution found
return BestSolution
```

The objective function to calculate the gain is the following:

$$objective = W_1\left(\sum_j utility_{c_j}(config\,params)\right) + W_2\left(\frac{bandwidth - BW}{BW}\right)$$
$$+W_3\left(\sum_d \frac{load_{sd} - CPUspeed_d}{CPUspeed_d}\right) + \sum_{constrainted\ m} W_4\left(\frac{T_{m_{sc_ic_j}} - threshold_m}{threshold_m}\right)$$

where the functions $W_i(x)$ are defined as:

$$W_i(x) = \begin{cases} w_i \times x & \text{if } x < 0 \\ 0 & \text{otherwise} \end{cases}$$

Thus, the objective function maximizes the utility, but adds in penalty factors weighted by w_i when the constraints are not met.

Note that also moves with a negative gain are performed when no better moves are found. This enables the heuristic to escape from local maxima. At the start of the inner loop, the $ExploredMoves$ list is also initialized with all moves that lead to the current solution ($InitialMoves$), in order to prevent the algorithm to get stuck in the current solution when a local optimum is found.

5 Experimental Results

5.1 AR Use Case

As a use case, we present an augmented reality application featuring markerless tracking as described by Klein et al. [9], combined with an object recognition algorithm presented in [11]. The application is shown in Figure 5. In the middle a greyscale video frame is shown with the tracked feature points, from which the camera position is estimated. The left part shows the resulting overlay with a 3D object, and a white border around the recognized book. On the right two mobile devices running the application are shown, forming a cloudlet with a laptop connected via WiFi.

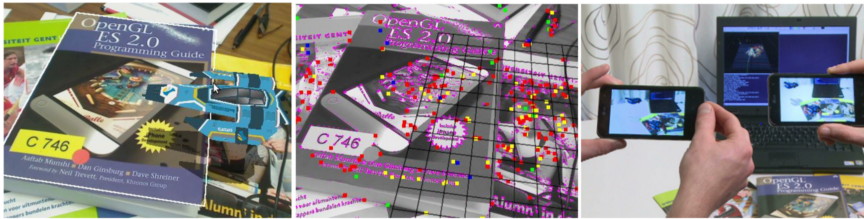


Fig. 5. The augmented reality application tracks feature points in the video frames (middle) to enable the overlay of 3D objects (left). Multiple mobile devices can run the same application while offloading components to a laptop in the cloudlet (right).

A component based implementation of this application was realized, and the three sequences shown in Figure 6 were identified. The first sequence (Fig. 6(a)) shows the tracking and rendering thread: the Video component periodically fetches a camera frame from the hardware, which is processed by the Tracker component. The tracker estimates the current camera position from tracked feature points, which is used by the Renderer to render the correct overlay. From

time to time the Tracker sends a video frame to the Mapper for map generation and refinement, which is shown in the second sequence (Fig. 6(b)) By matching 2D features in a sparse set of so called keyframes, the Mapper can estimate their 3D location in the scene and generate a 3D map of feature points. Finally, the keyframes are also analyzed for SIFT features, which are more complex to calculate, but can be used for object recognition by matching them against a database of SIFT features of known objects. This way objects can be recognized and localized in the map, which process is shown in third sequence (Fig. 6(c)).

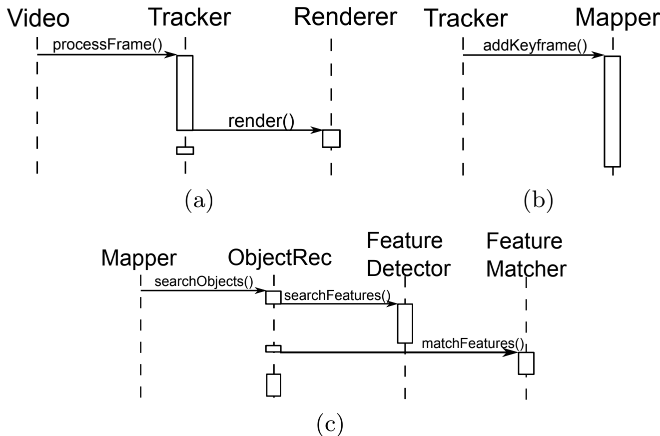


Fig. 6. The augmented reality application consists of three sequences. In (a) the tracking and rendering sequence is shown, which processes the video frames. The map refinement sequence is shown in (b), and (c) depicts the object recognition sequence.

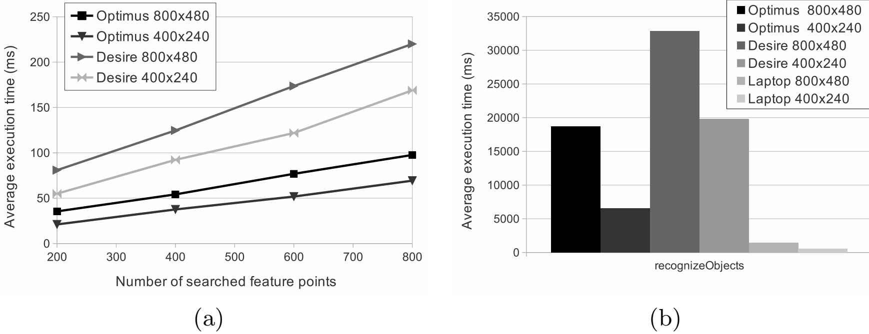
5.2 Results

We evaluated the AR use case on two mobile devices, forming a cloudlet together with a laptop connected via WiFi. The laptop is equipped with an Intel Core 2 Duo CPU clocked at 2.26GHz. As mobile devices we use a HTC Desire, with a single core Qualcomm 1 GHz Scorpion CPU, and an LG Optimus 2x powered by a dual core Nvidia Tegra 2 CPU, also clocked at 1GHz.

Two crucial configuration parameters affecting the application quality were identified: the camera resolution and the number of tracked features. Both devices support two resolutions: 800x480 and 400x240. The number of features to track affects the processing time of a frame by the Tracker (which is crucial to achieve an acceptable frame rate). Typical values for this parameter are 1000, 950, ..., 200. The more features tracked, the more robust the tracking, but the longer the processing time.

The monitored execution times of the tracker and object recognition sequences for different configurations are shown in Figure 7. Figure 7(a) shows that the

time to process a frame increases linearly with the number of feature points tracked. It also shows that the LG Optimus is 2 to 2.5 times faster than the HTC Desire. Figure 7(b) shows the processing times for object recognition, and again the Optimus is 2 to 3 times faster than the Desire, but the only acceptable processing times are achieved with the laptop, which is about 10 times faster than the Optimus. Therefore we set the relative $CPUspeed$ parameter as 0.4, 1 and 10 for the Desire, Optimus and laptop respectively.



(a) (b)

Fig. 7. Monitored execution times of the tracker (a) and object recognition (b) sequences, for different configurations

From the monitoring information we can set values for $Load_{m_{sc_ic_j}}$, $A_{m_{sc_ic_j}}$, $R_{m_{sc_ic_j}}$ for each method call. In this case each method call is executed only once in the sequence ($\#calls_{m_{sc_ic_j}} = 1$). Every five seconds one frame is added to the map and searched for objects ($\#calls_s = 0.2$). For the tracker sequence, the developer wants a minimal frame rate of 15 frames per second ($\#calls_s = 15$), meaning that a frame should be processed within 60ms, and objects should be recognized within 3 seconds. The devices are connected using a WiFi network of 10 Mbps and a latency of 1 ms.

Using this information, we can now calculate the optimal deployment and configuration. The Mapper, ObjectRecognizer, FeatureDetector and FeatureMatcher components are offloaded to the laptop. The Tracker components run on the mobile device, because of the limited bandwidth. Depending on the CPU capacity, the configuration is adapted to achieve the required frame rate. For the HTC Desire images are captured in 400x240 resolution and only 250 feature points are tracked, the Optimus captures frames in 800x480 resolution and tracks 500 points, as could be expected from Figure 7(a). The heuristic finds this result in 400ms, while a brute force implementation takes 16 minutes on the same hardware.

Figure 8 shows how the maximum achieved utility of the best solution varies as a function of the relative $CPUspeed$ of the device. The sudden increase around

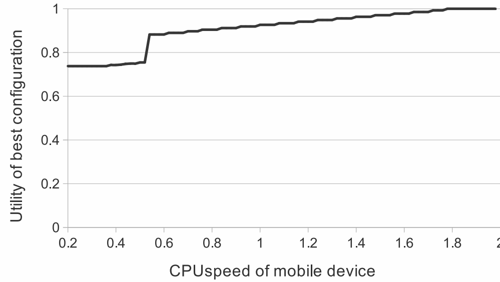


Fig. 8. The utility of the best possible configuration and deployment as a function of the devices $CPUspeed$

0.5 indicates the minimal $CPUspeed$ needed to process higher resolution frames. The small increments represent increases in the number of feature points tracked.

6 Conclusion

In this paper we present a cloudlet middleware architecture, that manages application on a component level. The middleware can both adapt the deployment and the configuration of the components at runtime, in order to optimize the offered quality of experience to the end user. We propose a decision algorithm that optimizes the application configuration and distribution, taking into account the network connectivity, the available resources and application constraints imposed by the application developer. Experimental results for a mobile augmented reality application show that the algorithm is indeed able to calculate the optimal solution, at a fraction of the time of a brute force implementation. Future work consists of further evaluating the quality of the heuristic, as well as integrating the algorithm in a full implementation of the cloudlet middleware.

Acknowledgment. Tim Verbelen is funded by Ph.D grant of the Fund for Scientific Research, Flanders (FWO-V).

References

1. Balan, R., Flinn, J., Satyanarayanan, M., Sinnamohideen, S., Yang, H.: The case for cyber foraging. In: EW 10: Proc. of the 10th Workshop on ACM SIGOPS European Workshop, pp. 87–92 (2002)
2. Chun, B., Ihm, S., Maniatis, P., Naik, M., Patti, A.: Clonecloud: elastic execution between mobile device and cloud. In: Proc. of the Sixth Conference on Computer Systems, EuroSys 2011, pp. 301–314 (2011)
3. Cuervo, E., Balasubramanian, A., Cho, D., Wolman, A., Saroiu, S., Chandra, R., Bahl, P.: Maui: making smartphones last longer with code offload. In: Proc. of the 8th International Conference on Mobile Systems, Applications, and Services, MobiSys 2010, pp. 49–62 (2010)

4. Gartner Group. 2012 press releases, http://www.gartner.com/it/page.jsp?id=2017015
5. Giurgiu, I., Riva, O., Juric, D., Krivulev, I., Alonso, G.: Calling the Cloud: Enabling Mobile Phones as Interfaces to Cloud Applications. In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 83–102. Springer, Heidelberg (2009)
6. Goyal, S., Carter, J.: A lightweight secure cyber foraging infrastructure for resource-constrained devices. In: WMCSA 2004: Proc. of the Sixth IEEE Workshop on Mobile Computing Systems and Applications, pp. 186–195 (2004)
7. Hassan, M.A., Chen, S.: An Investigation of Different Computing Sources for Mobile Application Outsourcing on the Road. In: Venkatasubramanian, N., Getov, V., Steglich, S. (eds.) Mobilware 2011. LNICST, vol. 93, pp. 153–166. Springer, Heidelberg (2012)
8. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. Bell System Technical Journal 49(2), 291–307 (1970)
9. Klein, G., Murray, D.: Parallel tracking and mapping for small ar workspaces. In: Proc. of the 6th IEEE and ACM International Symposium on Mixed and Augmented Reality, ISMAR 2007, pp. 1–10 (2007)
10. Kristensen, M.D.: Scavenger: Transparent development of efficient cyber foraging applications. In: 2010 IEEE International Conference on Pervasive Computing and Communications (PerCom), pp. 217–226 (2010)
11. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. Int. J. Comput. Vision 60(2), 91–110 (2004)
12. Ou, S., Yang, K., Zhang, J.: An effective offloading middleware for pervasive services on mobile devices. Pervasive and Mobile Computing 3(4), 362–385 (2007)
13. Satyanarayanan, M., Bahl, P., Caceres, R., Davies, N.: The case for vm-based cloudlets in mobile computing. IEEE Pervasive Computing 8(4), 14–23 (2009)
14. Su, Y., Flinn, J.: Slingshot: deploying stateful services in wireless hotspots. In: MobiSys 2005: Proc. of the 3rd International Conference on Mobile Systems, Applications, and Services, pp. 79–92 (2005)
15. Verbelen, T., Hens, R., Stevens, T., De Turck, F., Dhoedt, B.: Adaptive Online Deployment for Resource Constrained Mobile Smart Clients. In: Cai, Y., Magedanz, T., Li, M., Xia, J., Giannelli, C. (eds.) Mobilware 2010. LNICST, vol. 48, pp. 115–128. Springer, Heidelberg (2010)
16. Verbelen, T., Simoens, P., De Turck, F., Dhoedt, B.: Cloudlets: Bringing the cloud to the mobile user. In: Proc. of the 3rd ACM Workshop on Mobile Cloud Computing & Services, MCS 2012 (2012)
17. Zhang, X., Jeong, S., Kunjithapatham, A., Gibbs, S.: Towards an Elastic Application Model for Augmenting Computing Capabilities of Mobile Platforms. In: Cai, Y., Magedanz, T., Li, M., Xia, J., Giannelli, C. (eds.) Mobilware 2010. LNICST, vol. 48, pp. 161–174. Springer, Heidelberg (2010)