

# Dynamically Partitioning Applications between Weak Devices and Clouds

Byung-Gon Chun  
Intel Labs Berkeley

Petros Maniatis  
Intel Labs Berkeley

## ABSTRACT

Mobile cloud computing applications run diverse workloads under diverse device platforms, networks, and clouds. Traditionally these applications are statically partitioned between weak devices and clouds, thus may be significantly inefficient in heterogeneous environments and workloads. We introduce the notion of dynamic partitioning of applications between weak devices and clouds and argue that this is key to addressing heterogeneity problems. We formulate the dynamic partitioning problem and discuss major research challenges around system support for dynamic partitioning.

## 1. INTRODUCTION

In recent years, weak mobile devices such as smartphones, mobile Internet devices, and netbooks have become popular. Examples include the Apple iPhone [2], Google Android Phone [1, 7], RIM Blackberry Phone [3], and netbooks from several vendors. Tablets are also gaining attraction thanks to Apple's new iPad [6]. Moreover, weak devices are embedded in vehicles and TVs (e.g., Ford SYNC [4] and Google TV [5]).

These devices support exciting new applications that provide a connected Internet experience to users. Now users manage their Facebook and Twitter notifications; capture, edit, and upload photos and videos; handle their finances; and play computer games. Users also have growing demand for running more resource-demanding applications, the kinds of applications they run on laptops or desktops, such as rich media applications using diverse inputs like cameras and sensors. Examples include photo and video editing and synthesis, 3D modeling for constructing avatars, scene recognition, object recognition, image search, and augmented reality [12, 13, 18, 23].

Currently these applications running on weak devices are structured such that they are *statically* partitioned between the weak device and a server running in the cloud. Two representative application partitions are as follows. First, most of an application's processing is done at the server(s) in the

cloud, and the end-user device runs simple tasks such as UIs, acting like a thin client. For example, Facebook and Tweeter clients belong to this category. Second, most of an application's processing is performed at the client as exhibited in interactive, graphics-intensive games. These different partitions incur different costs (e.g., execution time, energy consumption, etc.).

However, partitioning applications statically does not provide optimal user experience as more and more applications are used in diverse environments and inputs. That is, there is no single partitioning that fits all due to environment heterogeneity (device, network, and cloud) and workload. Furthermore, there are many partitioning choices. For example, for an application that runs both on an Android phone and on an Android netbook, the application may run better in each device by using different partitioning between device and cloud. An Android phone application may run better in different network types by using different partitioning that incurs different computing and communication cost.

Our vision is a system that can seamlessly adapt to different environments and workloads by dynamically instantiating what partitioning to use between weak devices and clouds. In Section 2, we argue for the case for dynamic partitioning for different environments and workloads. Section 3 presents our formulation of dynamic partitioning problems, and Section 4 presents the system support for dynamic partitioning and key research challenges at a high level. We discuss related work in Section 5 and conclude in Section 6.

## 2. THE CASE FOR DYNAMIC PARTITIONING

To facilitate our discussion, we take an example application that performs image matching algorithms used for image search, object recognition, and augmented reality. Image matching consists of multiple processing stages in sequence — image feature generation, similarity calculation against a database, and classification. The feature generation stage such as SIFT [18] or SURF [8] is further composed of keypoint extraction, local computation to decide orientation, and descriptor computation. Similarity calculation is to compute minimum Euclidean distance for the the invariant descriptor vector of the image against the database of keypoints from training images. The similarity calculation and classification find an approximate nearest neighbor. The processing time of feature generation is data-dependent. Images with complex content take longer to process. Also, similarity computation takes longer to process as the database to search grows.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MCS June 15, 2010, San Francisco, CA, USA.

Copyright 2010 ACM 978-1-4503-0155-8 ...\$10.00.

The total execution time of image matching depends on multiple factors — device platforms, networks, clouds, and workloads. Therefore, the application has different optimal partitions for different environments and workloads. In the above examples, we have five steps and we have six different partitions to use. Instead of choosing one static partitioning, it would be ideal to choose what partitioning to use on demand depending on environments and workloads.

Next we delve into what heterogeneity problems we face in mobile cloud computing and how dynamic partitioning can address the problems incurred by the heterogeneity of environments and workloads.

### *Device Platform.*

In recent years we have seen diverse *weak* platforms running the same application: smartphones, mobile Internet devices, tablets, netbooks, car devices, and TVs tend to run the same application often downloaded from an application store. For example, the iPhone and iPad can run the same application downloaded from Apple’s AppStore; Google Android phone, Android-based netbook, and Google TV can run the same application downloaded from Android market; etc.

However, such devices have a wide variety of CPU, memory, and persistent store components. CPUs have different micro-architectures and frequencies, ranging from a 500MHz ARM processor without floating point unit to a 2GHz Atom x86 processor with floating point unit. In addition, some CPUs come with accelerators for signal processing. Memory also varies from 192MB to 2GB. The persistent store varies from 256MB to 64GB.

### *Network.*

The same application is used in different network types and conditions. 3G wireless networks supporting 2Mbps peak stationary and 384Kbps peak mobile bandwidth are becoming widely used, and there are WiFi hotspots at home, office, and public places like Starbucks’-cafes. Wimax, one of the 4G technologies, is being deployed and LTE is coming. LTE supports 1Gbps peak stationary and 100Mbps peak mobile bandwidth.

The network condition can also change depending on location. When the network may not be available (e.g., at a mountain), it would be ideal to run the application locally. However, when the network becomes available, the application can perform processing at the server in the cloud. The wireless link capacity can vary as well due to mobility and interference, hence this variability can change the application’s optimum partitioning.

### *Cloud.*

The application’s response time can change based on cloud conditions. Although we expect clouds to have high availability, they may not be available in some cases. The network distance to the cloud also affects the performance of the application. If one travels in Europe, should she still use a server in the United States? Cloud’s pricing can also be a concern. If the server runs in public clouds like Amazon’s EC2, the server’s capacity may change over time according to cloud prices, thus the application’s partitioning can adapt to consider the price factor. Dynamic pricing (e.g., Amazon Spot Instances) adds further flexibility to the partitioning problem.

### *Workload.*

Applications deal with different inputs with different sizes and different computational complexity. The computation time depends on the input content as well as the input size. Therefore, depending on workloads, it is ideal to dynamically instantiate what processing to do at the device and at the cloud.

### *Partitioning Scenarios.*

Optimal partitioning is determined by the (estimated) performance metric:  $Perf_{partitioned} = Exec_{weak} + Exec_{cloud} + Exec_{overhead}$ , characterizing the execution of the application partially on the weak device and partially on the cloud, along with the required execution overhead. The decision may be impacted not only by the application itself, but also by the expected workload and the execution conditions, such as network connectivity and CPU speeds of both weak and cloud devices, which affect all *Exec* terms. Traditional client-server partitionings hard-wire early on in the development process, but we need fine-grained flexibility on what to run where to address the heterogeneity problems. Next, we discuss a few partitioning scenarios for the image matching application in detail.

The image matching application can take advantage of more powerful processors and more memory at the device by running the first few steps (e.g., feature generation) locally, thus sending less data to the server. Also, if the application can exploit GPUs in the device, it can push more processing to the client. On the other hand, if the device’s CPU is slow and lacks the floating point unit, it may be better to send raw data to the server to perform all stages at the server.

The time to transmit data between client and cloud depends on the network speed, so the optimal partitioning of the image matching application between client and cloud can change. In 3G wireless networks, the image matching application may execute faster by running feature generation even though the device on which it operates has low computing power because it takes non-negligible time to send data over the network. In contrast, if the device has access to WiFi, the application may offload more processing to the server in the cloud.

Moreover, the application execution time depends on images. The computation time depends on image resolution, and it also depends on the data content itself. If the image matching application processes image(s) with simple objects (e.g., an apple in a white background wall), it may be faster to do feature generation at the device. However, if it processes image(s) with complex objects (e.g., a robot in cluttered backgrounds), it may be better to do more processing at the server. In addition, nearest neighbor processing depends on the database size. If the database size is small, it may be better to run the processing against the database locally.

## **3. PROBLEM FORMULATION**

We define an application as a collection of processing modules interacting with each other. For example, modules are functions, and when a function calls another function, the two functions interact. For a typical client-server application, both the client and server modules belong to the application. The client runs at the weak device, and the server runs in machines in the cloud. We define a partition between client and server as the two sets of modules that cover all

$$\begin{aligned}
& \underset{P_d, P_s}{\text{minimize}} && \sum_{m \in P_d} C_p(m, d, i) + \sum_{m \in P_s} C_p(m, s, i) + \sum_{m1 \in P_d, m2 \in P_s} C_c(m1, m2, t, i) \\
& \text{subject to} && m \in P_d, \forall m \in L_1(d) \\
& && m \in P_s, \forall m \in L_1(s) \\
& && m1, m2 \in P_d \text{ or } m1, m2 \in P_s, \forall (m1, m2) \in L_2
\end{aligned}$$

Figure 2: Formulation of the basic partitioning problem.

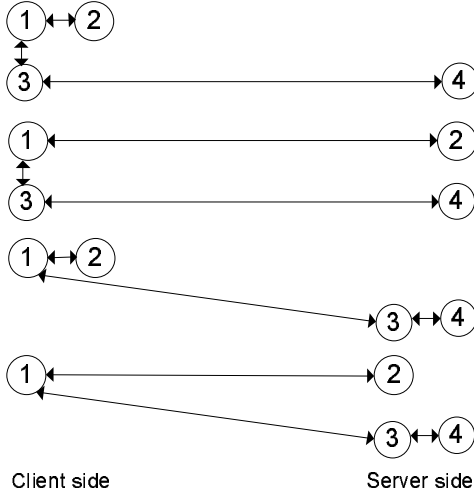


Figure 1: Examples of partitioning of an application composed of four modules when module 1 is pinned to the client and module 4 is pinned to the server. Note that even with this simple application there are four possible partitioning examples. Dynamic partitioning chooses what partitioning to use at run time among the set of available partitioning configurations.

modules of the application. Formally, we define  $A$  as the entire set of modules, define  $P_d$  as the partition for mobile device  $d$ , and define  $P_s$  as the partition for the server  $s$  running in the cloud;  $P_d \cup P_s = A$ . Figure 1 shows partitioning examples. A circle represents a module, and an edge represents that two modules connected by the edge interact. In our discussion we focus on the basic partitioning case where two non-overlapping sets cover the entire application:  $P_d \cup P_s = A$  and  $P_d \cap P_s = \{\}$ . Note that there are more complex cases like partitioning between a client and multiple servers and partitioning with overlapping sets of modules (e.g., the same module executes at the client and the server).

We can cast this partitioning problem as an optimization problem. To define the problem, we need to define optimization goals with cost functions and optimization constraints. The optimization goal can be to minimize execution time, minimize battery power consumption if the device is not plugged into the power grid, or a combination of both. There are constraints in the optimization problem. Some modules need to be pinned at a particular location. For example, modules accessing a hardware feature (e.g., device camera) should be located at the machine with the feature in any valid partition. Or two modules that may have dependencies in the state of the same hardware device (e.g., frame buffer

in graphics units) need to be located at the same machine.

Now we present a formal problem to optimize the total cost (e.g., the total execution time) of an application composed of modules. We define  $C_p(m, l, i)$  as the cost of processing module  $m$  at location  $l$  for input  $i$ . The cost of processing module  $m$  depends on the platform running at  $l$ . As we discussed in Section 2, different device platforms with different CPU, memory, persistent store, and hardware features incur different cost to execute  $m$ . We define  $C_c(m1, m2, t, i)$  as the cost of communication between module  $m1$  and module  $m2$  over network  $t$  for input  $i$ . Different network types and conditions affect this communication cost. If module  $m$  makes transactions frequently to backend database servers, the communication cost of putting  $m$  at the device is high. We denote  $L_1(l)$  as the set of modules that need to be pinned to location  $l$ , and denote  $L_2$  as the set of pairs of modules that need to be colocated. The optimization problem for given input  $i$ , network type  $t$ , device  $d$ , and server  $s$  is shown in Figure 2.

## 4. SYSTEM SUPPORT FOR DYNAMIC PARTITIONING

Dynamic partitioning chooses and executes partitions of the application that optimize the application’s goal we formulated above at run time. To achieve this vision, we need system support for dynamic partitioning. In the following, we summarize three major research challenges: structuring applications, choosing what partitioning to use, and partitioning applications with security constraints.

### Application Structuring.

To enable dynamic partitioning, we first have to architect programs to execute partitions seamlessly and dynamically between the weak device and the cloud. This requires different application structuring from that used traditionally. First, since a module may be executed either at the client or at the server, both client and server must have all parts of the application. If there is any change in a module due to a software upgrade, both client and server must be updated. Second, the application needs to instantiate what modules to run at the client and at the server dynamically at run time. When the modules are located at the same machine, they can interact through local procedure calls or through an inter-process communication, both optimized for processes running in the same machine. When the modules are located at different machines, the dynamic partitioning execution system needs to connect these modules through remote procedure calls or migration mechanisms seamlessly. Unlike static partitioning, the wiring of the modules occurs at run time and can change over time in dynamic partitioning. This flexible wiring is a key component of dynamic partitioning

execution.

### Partitioning Choice.

Once we have a mechanism to seamlessly execute distributed modules of the application, we need to choose a partitioning configuration to use at run time. Like many problems in systems, we can look at this problem in two parts: policy and mechanism.

Where to place modules depends on what to optimize. The optimization goals may be execution time, battery power consumption, or the total money spent to execute the application. In some cases, the goals may capture the security impact. If the user has a strong preference towards privacy, the application can run more modules at the client side to preserve privacy (e.g., targeted advertising systems like PrivAd [15] and Adnostic [24]).

Partitioning decision also depends on what the input is. The optimization problem presented in Figure 2 depends on input  $i$ . Collecting a right set of inputs is crucial for optimal partitioning because we need to create a model from the analysis of the optimization problem (e.g., a function that maps inputs and environments to partitions).

The mechanism to choose which partitioning to use at run time must be lightweight since it runs at the weak device. Ideally, from the collected measurements of  $C_p$ 's and  $C_c$ 's, we would like to solve the optimization problem shown in Figure 2, but it is not realistic to try to solve this complex optimization problem right before running the application. There are two subproblems: 1) how to know the costs before actually running the application for the given input and environment, and 2) how to choose the best partitioning. One approach to address these problems is to create a prediction model through offline analysis. The model can predict costs of different partitioning configurations before running the application. The client can then use a simple approximate optimizer that can run quickly based on the predicted costs (e.g., relaxing an ILP problem to an LP problem to trade accuracy for speed).

### Security of Dynamic Partitioning.

Finally, there are security issues to address — privacy, data confidentiality and integrity. In static partitioning, a module is fixed to run either at the client or at the server. If a module contains sensitive server-side data, the module should run at the server. In dynamic partitioning, we need to make sure that a module containing the sensitive data of a machine does not run at another machine. This requirement brings up challenging privacy-preserving partitioning problems. There are approaches like Swift [9] that rely on programmer annotations. One interesting research direction is to look at more automated approaches of privacy preserving partitioning through static and dynamic program analysis.

## 5. RELATED WORK

We briefly summarize previous work on the partitioning of distributed systems. This work focuses on static partitioning of applications. In contrast, we focus on dynamic partitioning of applications between client and server in heterogeneous workloads and environments and sketch the system support needed for dynamic partitioning.

There has been research on partitioning a program into client and server parts such as Links [11], Hops [22], and

UML-based Hilda [26]. The program is written in a high-level functional language or high-level declarative language; the source code is *statically* partitioned into two or three tiers. Yang et. al [25] examine partitioning of programs written in Hilda based on cost functions for optimizing user response time.

Coign [16] automatically partitions a distributed application composed of Microsoft COM components to reduce communication cost of partitioned components. Wishbone [20] is a system that takes an acyclic dataflow graph of operators written in a high-level stream-processing language and partitions the dataflow graph between server and a set of embedded nodes for sensor computing applications. It uses a compiler that generates partitioned source code with communication stubs based on profiling CPU and network bandwidth consumption. Pleiades [17] compiles a central program written in an extended C language with the model of accessing the entire network into multiple units to run on sensor nodes.

To offload some parts of the client application, Java program partitioning for mobile devices has been explored with limitations [14, 19, 21]; only Java classes without native state can be placed remotely. The general approach is to partition Java classes into groups using adapted MINCUT heuristic algorithms to keep the component interactions between partitions as small as possible. Also, different proposals consider different additional objectives such as memory, CPU, or bandwidth. CloneCloud [10], our own proposal, creates clone(s) of mobile devices and offloads some part of the application to the clone(s) running in the cloud. The focus is more on augmenting current mobile device applications with the cloud.

Finally, Swift [9] statically partitions a program written in the Jif programming language into client-side and server-side computation. Its focus is to achieve confidentiality and integrity of the partitioned program with the help of security labels in the program annotated by programmers.

## 6. CONCLUSION

In this paper, we argue for dynamic partitioning of applications between weak device and cloud to better support applications running in diverse devices in different environments. We point out different sources of heterogeneity and discuss how dynamic partitioning can address these heterogeneity problems. We then formalize the dynamic partitioning problem, and sketch how to construct a system that supports dynamic partitioning. We believe that dynamic partitioning is an important part of future mobile cloud computing.

## 7. ACKNOWLEDGMENTS

We thank Ling Huang for the discussion of image matching applications.

## 8. REFERENCES

- [1] Android dev phone 1. [code.google.com/android/dev-devices.html](http://code.google.com/android/dev-devices.html).
- [2] Apple iPhone. [www.apple.com/iphone](http://www.apple.com/iphone).
- [3] Blackberry smart phones. [na.blackberry.com/eng/](http://na.blackberry.com/eng/).
- [4] Ford SYNC. [www.fordvehicles.com/innovation/sync](http://www.fordvehicles.com/innovation/sync).

- [5] Google TV. [www.nytimes.com/2010/03/18/technology/18webtv.html?partner=rssnyt&emc=rss](http://www.nytimes.com/2010/03/18/technology/18webtv.html?partner=rssnyt&emc=rss).
- [6] iPad. [www.apple.com/ipad](http://www.apple.com/ipad).
- [7] Nexus One. [www.google.com/phone](http://www.google.com/phone).
- [8] H. Bay, T. Tuytelaars, and L. V. Gool. SURF: Speeded up robust features. In *ECCV International Workshop on Statistical Learning in Computer Vision*, 2006.
- [9] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure Web Applications via Automatic Partitioning. In *SOSP*, 2007.
- [10] B.-G. Chun and P. Maniatis. Augmented Smartphone Applications through Clone Cloud Execution. In *HotOS*, 2009.
- [11] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Proc. 5th International Symposium on Formal Methods for Components and Objects*, 2006.
- [12] C. Dance, J. Willamowski, L. Fan, C. Bray, and G. Csurka. Visual categorization with bags of keypoints. In *ECCV International Workshop on Statistical Learning in Computer Vision*, 2004.
- [13] R. Fergus, P. Perona, and A. Zisserman. Object class recognition by unsupervised scale-invariant learning. In *IEEE Conf. on Computer Vision and Pattern Recognition*, 2003.
- [14] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic. Adaptive Offloading Inference for Delivering Applications in Pervasive Computing Environments. In *PERCOM*, 2003.
- [15] S. Guha, A. Reznichenko, H. Haddadi, and P. Francis. Serving ads from localhost for performance, privacy, and profit. In *HotNets*, 2009.
- [16] G. C. Hunt and M. L. Scott. The coign automatic distributed partitioning system. In *OSDI*, 1999.
- [17] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *PLDI*, 2007.
- [18] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Journal of Computer Vision*, 2004.
- [19] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T. Giuli, and X. Gu. Towards a distributed platform for resource-constrained devices. In *IEEE ICDCS*, 2002.
- [20] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Profile-based partitioning for sensornet applications. In *NSDI*, 2009.
- [21] S. Ou, K. Yang, and J. Zhang. An effective offloading middleware for pervasive services on mobile devices. *Pervasive Mob. Comput.*, 3(4), 2007.
- [22] M. Serrano, E. Gallesio, and F. Loitsch. HOP: a language for programming the web 2.0. In *Proc. 1st Dynamic Languages Symposium*, 2006.
- [23] N. Snavely, S. M. Seitz, and R. Szeliski. Photo tourism: Exploring photo collections. *ACM Trans. on Graphics*, 2006.
- [24] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *NDSS*, 2010.
- [25] F. Yang, N. Gupta, N. Gerner, X. Qi, A. Demers, J. Gehrke, and J. Shanmugasundaram. A unified platform for data driven web applications with automatic client-server partitioning. In *WWW*, 2007.
- [26] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-driven web application. In *ICDE*, 2006.