Automated Component Insourcing for Mobile Applications

Undoing Premature or Ill-Conceived Offloading Optimizations

Eli Tilevich

Virginia Tech

tilevich@cs.vt.edu

Abstract

To alleviate the resource constraints of mobile devices, developers of mobile applications commonly apply the cloud offloading optimization—placing an application's energy and performance intensive functionality to execute at a remote cloud server. Because of the high heterogeneity of mobile hardware and the variability of mobile networks, a cloud offloading optimization may become detrimental to energy consumption and performance efficiency. Reverting to the original, non-distributed version of the application may be infeasible, as the client and server parts may since have been maintained and enhanced independently. In that case, the components executed at the server for optimization purposes need to be moved to be executed on the mobile device; the moved components must be integrated with the rest of the application's functionality. To assist software developers with this non-trivial program restructuring, this paper introduces *component insourcing*, an automated program transformation that moves a remotely accessed component to be invoked locally, within a shared address space. This position paper motivates the need for component insourcing and demonstrates how this transformation can be implemented in the Java realm. The paper also discusses the technical difficulties of preserving the original component behavior and ensuring good performance in the presence of component insourcing.

Categories and Subject Descriptors D.2.7 [*Distribution, Maintenance, and Enhancement*]: Restructuring, reverse engineering, and reengineering

Keywords mobile applications, cloud offloading, program transformation, optimization

1. Introduction

As mobile devices are rapidly overtaking stationary computers as the primary means of accessing computing resources, battery capacities remain a key physical constraint of mobile execution [6]. As a means of reducing the energy consumption of mobile devices, developers of mobile applications commonly leverage distributed execution, in which energy and execution intensive functionality is placed to be executed at a cloud server and to be accessed remotely across the network. Removing energy intensive functionality from the code running on a mobile device reduces the execution demands on the device's battery, thus lengthening its life. Once software developers carry out a cloud offloading optimization, the mobile application is then executed in a distributed fashion. The client and server parts are then free to be evolved independently, as long as the communication protocol between them remains unaffected.

An offloading optimization may turn out premature or ill-conceived for two reasons. First, the mobile hardware market is highly heterogeneous: according to Facebook, the mobile version of their application is accessed from 2,500 varieties of mobile devices [3], with each device having different energy consumption characteristics. A given cloud offloading optimization is likely to exhibit different levels of effectiveness depending on the mobile device in use. Second, the network conditions in place can have a profound effect on how much energy a mobile device spends on transferring the same workload to a remote server [5].

As a specific example, consider a mobile application being executed on a mobile device plugged to a power source and connected to the server by means of a cellular network with a low signal strength. This scenario could occur when a mobile device is plugged to the charger of a car traveling through a rural area. An offloading optimization applied to this application is likely to deteriorate its performance if not render it unusable altogether. As another example, running an application optimized for energy efficient execution on a smartphone is unlikely to yield comparable energy savings when run on a tablet.

These scenarios motivate the need for moving some distributed components from a remote server to the mobile device, thereby undoing cloud offloading optimizations that stopped being effective. With different mobile hardware, the application's non-distributed version may turn to be sufficiently energy efficient or can be used as a basis for an alternate offloading optimization. For example, some parallel computations can be offloaded to a newly introduced GPGPU with comparable energy savings. However, moving remote functionality to the mobile device to be accessed locally entails non-trivial program transformations, particularly if the resulting "glued" application is to exhibit good performance.

To alleviate the burden of undoing cloud offloading optimizations, this paper introduces *component insourcing*, an automated program transformation that adapts a remote component to be accessed locally within a shared address space. To preserve the original behavior of remote components when executed locally, component insourcing must properly reconcile the differences in parameter passing semantics, failure modes, and middleware dependencies between the remote and local execution models. A crucial requirement of component insourcing is ensuring the ability to access the formerly remote functionality locally without suffering a prohibitive performance overhead. Addressing this requirement requires novel solutions to a set of technical problems that this position paper defines in the subsequent discussion.

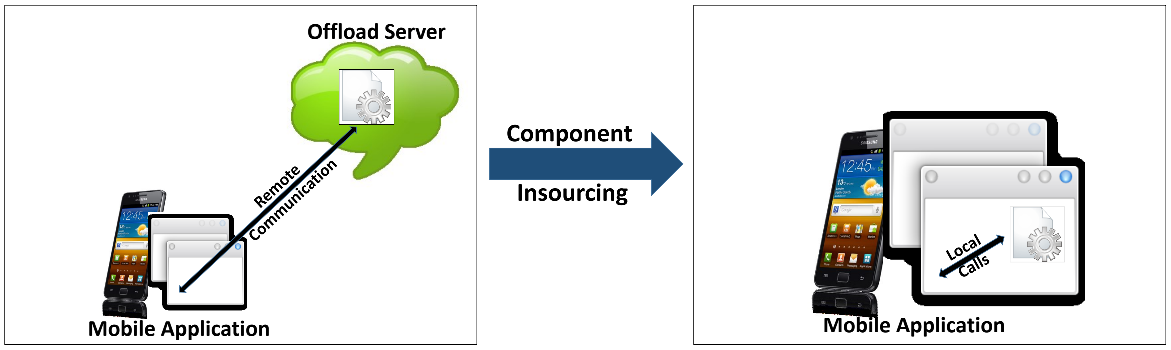

Figure 1. The Component Insourcing Program Transformation

2. Component Insourcing

Figure 1 shows *component insourcing*, a program transformation that moves a remote server component to the client's address space and replaces remote communication with local calls. By *remote*, we mean executing in a separate address space, usually at a different network node, such as at a cloud server. In essence, component insourcing undoes distribution decisions made when either initially designing a mobile application or optimizing its energy efficiency. Obviously, not all distributed components can be successfully insourced, as explained next.

2.1 Applicability

To understand the applicability of component insourcing, one must ask why distributed execution is introduced in the first place. To date, distributed processing is used for two primary reasons: (1) *functional distribution*—accessing a remote functionality not available locally, and (2) *distribution to improve performance*—leveraging distributed computing resources to efficiently execute computationally intensive tasks. Although it may seem that these reasons behind distributed processing may not be mutually exclusive, there is one characteristic that clearly sets them apart. While functional distribution is inevitable, distribution to improve performance can be replaced with equivalent, albeit less efficient local computation.

As an example of functional distribution, consider a mobile application for accessing the content of a newspaper. The application uses distributed processing not because accessing the newspaper from a remote server is faster. The distributed communication is necessary to retrieve the newspaper's articles from a remote server (or a server farm). The issue of execution performance is secondary when analyzing this application's distributed architecture. Without accessing the remote content, the application would be quite useless. The distributed execution and communication must take place even if the remote server's performance is suboptimal or the cellular network in place has a poor signal. Even if the mobile device had unlimited processing resources, no amount of local execution can generate the newspaper's content locally.

As an example of distribution to improve performance, consider using a remote optical character recognition (OCR) component. OCR algorithms take a graphical image as input and decipher the letters comprising the image. These algorithms are known to be a highly execution intensive operation, and as such likely to exhaust the battery power of a mobile device quickly. Therefore, a mobile application designer may want to take advantage of a remote server to execute the OCR functionality. The energy costs of sending an image file to the remote components and getting the textual output back is likely to be lower than executing the OCR algorithm locally. However, this optimization is optional. For example, if the application runs on a mobile device connected to a power source, the issue of preserving the battery power is quite irrelevant. Similarly, in the presence of a highly unreliable cellular network, the costs of handling network disconnections are likely to outstrip those afforded by offloading the OCR computation to the remote server. All in all, the OCR functionality can be implemented as either a remote or local component, depending on the energy efficiency and QoS requirements in place.

3. Challenges of Component Insourcing

To demonstrate the technical issues underlying component insourcing, we will use the following running example. For ease of presentation, this example assumes that both client-side and server-side software is written in Java; it also assumes that the client communicates with the server via Java Remote Method Invocation (RMI). These are simplifying assumptions, but insourcing components written in other languages and removing the harness of additional middleware infrastructures are primarily engineering issues that should not affect the conceptual technical issues discussed next.

3.1 Running Example

Consider a server-side component, whose entry point is located in class Component as shown in Figure 2. One of its remote methods on line 3 takes as a parameter a Set of elements. The method processes the Item objects contained in the parameter and returns the calculated Result object back to the client.

```
 1  class Component implements RemoteInterface {
 2    Result calculate (Set <Item> items)
 3                        throws Remote Exception {
 4      ...
 5      items.set(1, new Item (...));
 6      ...
 7      // use the passed items
 8      // to calculate and return the results
 9      return results ;
10    }
11  }
```

Figure 2. A server component to be insourced

Figure 3 shows the client code that first looks up a remote reference to a Component object from the registry service (line 5). The reference is assigned to the interface type of RemoteInterface, a common convention for invoking remote services in Java. This interface is then used to invoke the remote method calculate. The execution takes place at the server, and all the exceptions raised during the remote invocations are caught and handled on line 9.

```
 1 // create and populate the items object
 2 Set <Item> items = ...
 3 try {
 4   // lookup a remote reference from the registry
 5   RemoteInterface ri = Registry.lookup (...);
 6   Result r = ri.calculate (items);
 7   // use the Result object
 8 } catch (RemoteException re) {
 9     ... // handle remote service exceptions
10 }
```

Figure 3. A client code invoking the remote Component.

It may seem that insourcing this component to invoke it locally is as straightforward as the code that appears in Figure 4. In essence, this code instantiates a Component object directly and then proceeds to invoked its calculate method.

```
// Invoking the insourced Component locally
Component c = new Component();
Set <Item> items = ...
Result r = c.calculate (items);
```

Figure 4. Invoking the remote Component locally.

However, this transformation does not preserve the original execution semantics of interacting with the Component object remotely. In other words, this transformation is not a refactoring. To make this transformation semantics-preserving at least with respect to the application logic, the transformation must account for the following differences between the remote and local execution.

3.2 Parameter Passing

While remote calls use the `by-copy` semantics for passing parameters, local calls in managed languages use the `by-reference-value` semantics. Consider line 5 in Figure 2. The methods's parameter—Set of items is modified. This change would not be visible to the caller if method calculate is remote, as items is passed `by-copy` to the server, with no changes propagated back to the client once the call completes. When the method calculate is invoked locally, all the modifications to the items parameter will be reflected on the caller's reference to the items object. Parameter modifications like this can be subtle and hard to detect, but they change the application's semantics in major ways.

The same observation applies to non-primitive return values of remote methods. The return results object is copied back to the caller, which is free to modify its state. However, if the server has aliased this object prior to returning it to the caller, the client-site modification will not be reflected in the remote component's state. Once the component is insourced, the caller and callee will run in the same shared address space, in which the changes to an object are seen by all its aliases.

A naïve approach to emulating the `by-copy` semantics in local method calls is to deep copy the parameter before passing it to the method. Deep copying emulates the process of marshaling/unmarshaling the parameters of remote methods. The object graph of a parameter is traversed exhaustively and written to a memory buffer, then the buffer is transformed into an object graph isomorphic to the original parameter's graph. As a result, deep copying object graphs is a performance and energy intensive operation that should be used sparingly in mobile applications. To avoid the performance and energy consumption penalty, a more efficient approach to emulating the `by-copy` semantics is required.

3.3 Removing Middleware Harness

Remote components are typically managed by an *Inversion of Control (Ioc)* container. Container-based designs enable the separation of concerns principle, in which the container is responsible for concerns that include network communication, fault tolerance, and load balancing, while the components managed by the container only implement application (i.e., business) logic. To interact with container services, managed components commonly include special functionality. For example, RMI components have to implement a Remote interface, and each remotely invoked method must declare as throwing a RemoteException. It must be noted that compared to widely used middleware infrastructures for managing remote components, RMI imposes quite a limited set of requirements on the implementation conventions of its remote components. To adhere to the requirements of Web services, Java Services, and OSGi, remote components often need to provide additional methods and fields used exclusively by the container.

Accessing a component written to run within a container directly may not preserve the original semantics or may even cause a crash. Instantiating a container-managed component may involve protocols different than just invoking a constructor. For example, a call to the default constructor may be followed by that to an initialization method to allocate the component's resources. A straightforward but inefficient approach for executing insourced container-managed components is to continue running them within a container. However, running a container, designed for resource-rich server environments, on a mobile device will unnecessarily drain the scarce energy resources. Thus, when insourcing a component, its middleware functionality must be removed, so that the component can be efficiently invoked locally.

3.4 Handling Failure

Local and remote executions have radically different failure modes. Remote execution is subject to partial failure, in which each of the execution's constituent parts may fail separately. Handling partial failure effectively is one of the foremost challenges of distributed computing. In addition, by executing in a separate address space on a different machine, distributed components raise faults that do not affect their clients. Container-based distributed frameworks are particularly adept at handling such failures. For example, if a distributed component crashes during the execution, the container can restart it, with the client experiencing only an increased remote interaction latency.

When a remote component is insourced, its faults become visible to the client, as the isolation provided by executing in a separate address space is no longer present. One can execute insourced components in a separate address space, but modern languages provide little support for efficient multi-process execution. For example, although there were proposals to extend Java with Isolates, the JVM does not support efficient multiprocessing. Thus, to reliably execute the insourced components within a shared address space, the program transformation must insert the required failure handling code to guard the clients from the faults raised during the components' execution.

4. Supporting Component Insourcing

In this section, we outline some possible solutions to the technical challenges that must be addressed to realize the vision of component insourcing.

4.1 Parameter Passing

To efficiently emulate the `by-copy` semantics in a shared address space, only the modified subgraph of method parameters should be deep copied. To that end, a program transformation can rewrite the classes of the parameters passed to remote methods to keep track of all the field writes and reads. Once a field is about to be written, then only the object subgraph rooted in that field should be deep copied, and the copy written to instead. Arrays would need to be handled specially, with array stores treated as write operations. An additional optimization of not deep copying until the modified field is read is possible as well. Under some scenarios, a field can be modified but never written. Then the deep copy operation can be avoided altogether.

4.2 Removing Middleware Harness

The transformations required to remove the unnecessary middleware-related functionality would be defined by the programming conventions used by the middleware infrastructure in place. In addition, certain functionalities provided by the IoC container may need to be emulated when invoking the insourced components directly. For example, if an insourced component has a method `initialize` used to be invoked by the container, the calls to this method need to be added to the component's constructors. Similarly if a component has an `uninitialize` method, it may need be invoked explicitly or by means of `Runtime.addShutdownHook`.

4.3 Handling Failure

Maintaining the original failure handling semantics has to be addressed from the perspectives of remote communication and application faults. Because no partial failure can occur when invoking insourced components, one can safely remove all the code that handles network-related exceptions. However, to emulate the fault isolation provided by the execution in a separate address space can be non-trivial. If a crashed remote component can be simply restarted by its container, in a shared address space, the crash can bring down the entire virtual machine. One solution is to analyze the component's code for possible application failures and insert the appropriate fault handlers close to the potential causes of the fault to prevent them from propagating and creating a destructive ripple effect.

4.4 Automated Tools for Component Insourcing

Because component insourcing is essentially a refactoring, it can be supported by extending existing refactoring browsers. The proposed transformations described above can be added to an existing refactoring browser by leveraging its existing transformations, source code analyzers, and graphical interface. The removing middleware harness transformation can be particularly tricky because of its domain-specific nature.

5. Related Work

In essence, component insourcing undoes program partitioning, an automated program transformation that introduces distribution to a centralized application [7]. Although the bulk of the original research on program partitioning for managed languages took place during the first decade of this century, recently this technique reemerged under the guise of cloud offloading, a technique for increasing the energy efficiency of mobile applications [1, 2, 4, 8, 9]. Our vision is that offloading and insourcing transformations should be developed in concert with each other as complementary operations. Because the constituent parts of a distributed application can be evolved separately (with the client and server parts being individually maintained and enhanced), simply reverting back to the original, centralized version of the application may be infeasible, thus requiring a component insourcing transformation.

6. Conclusions

The realities of the modern mobile marketplace often require software developers to design and implement applications that are to be executed on a variety of mobile devices, some of which will be introduced only in the future. Because optimizing mobile applications for energy and performance efficiency is hard, cloud offloading optimizations may turn out premature or ill-conceived. As a result, mobile software engineers may find themselves undoing and redoing cloud offloading optimizations multiple times as mobile applications are maintained and evolved. Distributed components may also need to be insourced for security or privace reasons. For example, a mobile application mostly executed in untrustworthy environments should have its remote communication minimized.

Because of the rapid evolution of mobile devices and the tremendous growth in mobile applications, software engineers find themselves spending an increasing amount of their time and efforts on optimizing mobile execution for different devices and execution environments. Distributed execution is known to provide multiple benefits to mobile application, and various software development techniques and tools have been introduced to ease the process of transforming local components to be accessed remotely across the network. No techniques or tools have been introduced to go in the opposite direction: rendering remote components to be accessed locally within a shared address space. To address this problem, this paper introduces component insourcing and thus makes the following two contributions. First, it motivates and defines component insourcing, thus coining the term that describes this automated program transformation. Second, it identifies the research agenda for realizing the vision of component insourcing by describing its technical challenges.

Acknowledgments

This research is supported by the National Science Foundation through the Grant CCF-1116565.

References

[1] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: elastic execution between mobile device and cloud. In *Proceedings of the 6^{th} ACM European Conference on Computer Systems*, 2011.

[2] E. Cuervo, A. Balasubramanian, D.-K. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: making smartphones last longer with code offload. In *Proceedings of the 8^{th} International Conference on Mobile Systems, Applications, and Services*, 2010.

[3] Facebook Mobile. Facebook for every phone, July 2011.

[4] Y.-W. Kwon and E. Tilevich. Energy-efficient and fault-tolerant distributed mobile execution. In *Proceedings of the 32^{nd} International Conference on Distributed Computing Systems*, 2012.

[5] Y.-W. Kwon and E. Tilevich. The impact of distributed programming abstractions on application energy consumption. *Information and Software Technology*, 2013.

[6] K. Pentikousis. In search of energy-efficient mobile networking. *Communications Magazine, IEEE*, 48(1):95–103, 2010.

[7] E. Tilevich and Y. Smaragdakis. J-Orchestra: Enhancing Java programs with distribution capabilities. *ACM Trans. Softw. Eng. Methodol.*, 19(1):1–40, 2009.

[8] Y. Wen, W. Zhang, and H. Luo. Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones. In *Proceedings of the 32^{nd} IEEE Computer and Communications (INFOCOM '12)*, 2012.

[9] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang. Refactoring Android Java code for on-demand computation offloading. In *Proceedings of the ACM international conference on object oriented programming systems languages and applications*, OOPSLA '12, 2012.