Intelligent Mobile Code Offloading Decisions

Aluri, Ajit
aaluri@umich.edu

Arora, Silky
silkyar@umich.edu

Rao, Supriya
supriyar@umich.edu

Abstract

1. Introduction

Offloading mobile applications to the cloud helps to bridge the gap between the computing capabilities of mobile devices and server machines. With the advance of ubiquitous computing many low-powered computing devices can benefit from the high processing powers of available server machines. The elastic computational cloud fuctionality helps in augmenting the processing speeds on demand and also leads to power efficient execution of applications.

Mobile code offloading may involve analyzing the mobile application statically and partitioning it to identify the most computationally expensive operations. These operations can be run on a remote server when the client is well connected to the server. Theoritically, offloading is advisable in scenarios where the network conditions are good for offloading. Moreover, the mobile application should preferably have high amounts of computational processing which can execute much faster remotely. A good offloading engine should also take into account the amount of data that needs to be transfered to the server. Based on the current network conditions and the size of migration data, the offloading engine should make its decision. Thus a good offloading mechanism should take into account various network and application parameters depending on the approach used to implement offloading. Our main goal in this project is to improve this offloading decision for an already existing offloading infrastructure.

The backbone for our offloading decision is the COMET [4] infrastructure. COMET has built its offloading engine on the Dalvik Virtual Machine. COMET uses Distributed Shared Memory (DSM) to implement offloading. It can be used to offload any android application in general as the application need to contain any offloading logic. By using a DSM based approach, the offloading mechanism can support multi-threaded computation and allow threads to move between the client and the server. The current scheduling algorithm in COMET is based on a simple heuristic and can be improved. Moreover, COMET aborts the execution at the server when a native method is encountered. In an application with plenty of native calls, continuous abortion at the server might lead to performance loss as time is wasted in the data transfer and sync between client and server. To address these limitations, we use static analysis to devise heuristics that determine when methods should be offloaded.

Our primary contributions through this project are the following

- We perform static analysis of applications to identify native methods and their parent methods
- We compare the data transfer during sync operations to the current network bandwidth to make offloading decision
- Based on method execution times, we implement two offloading heuristics that identify the methods which can be marked as safe or unsafe for migration

2. Background and Motivation

In COMET (Code Offlaod by Migrating Execution Transparently) [4] the authors introduced a runtime system that allows migration of threads between systems depending on their workload. COMET extended distributed shared memory (DSM) techniques to enable offloading, a process in which a resource constrained device can execute compute intensive tasks on a relatively high performance device.

The core idea behind COMET is to manage memory consistency across different points at a field granularity using a VM Synchronization primitve. Changes to objects are tracked at a field level by setting an addtional dirty bit to true if the field was modified by an end point. The *happens-before* ordering between reads and writes of different threads helps in merging the changed fields to achieve a consistent state. A VM synchronization primitive is used to transfer the states of virtual heaps, stacks, source code (dex files) and synthetic classes to the other end point.

A VM sync is usually initiated when a thread requests for ownership of an object it doesn't own or when the thread can be migrated to the other end point for execution. The thread that initiated the sync starts collecting all the fields that have been changed since last sync. Once the dirtied fields have been obtained, the stacks that have to be synced are gathered and all the state changes are then transmitted over to the other end point. During changed state collection all threads are temporarily suspended and are resumed just before the data transfer is initiated. To avoid exchanging the code that the other end point has to resume from, all the source code files are transferred only on the first syncronization. The current program counter value gives the next instruction the other end point begins execution from. To support locking primitives, COMET assigns an owner to each lock (through a flag) indicating the end point that currently owns the object. When a thread encounters an object it doesn't own, an ownership transfer request is initiated to acquire the object, which could

result in a VM synchronization.

3. Intelligent Code Offloading

In the previous section, we identified certain limitations in the current offloading strategies in the COMET infrastructure. To address them we devised certain heuristics based on our analysis of android applications.

3.1. Parent-to-Native Execution Time

Currently, COMET identifies a set of native methods that can be offloaded. These methods are annotated manually and the run-time system determines if the current native method can be flagged as offloadable. This is done for approximately 200 native methods. In addition to these methods, the application has many other native function calls that cannot run on the server. The current offload infrastructure will abort execution at the server and invoke a sync call to send the state back to the client. This process of sending state from client to server and receiving state on aborting at native call can be very expensive, especially with limited network bandwidth.
We address this issue by profiling the application by obtaining the execution time betweeen the each parent method and its first native method invocation. If this execution time is very small as compared to the time required to send the data during a sync operation, then it would be unadvisable to perform a sync at the parent method and send the state to the server. By marking the parent as unsafe to offload, we avoid the sync time in sending and receiving data when the application aborts at the server. Figure 1 shows an example of a parent method Pa and its native method Na. t1 denotes the end of Pa and t2 denotes the start of the native method Na. The time duration $t2-t1$ includes the non-native execution code. We can compare this time difference with the last sync time (or get the sync time based on current bandwidth and data transfer size)

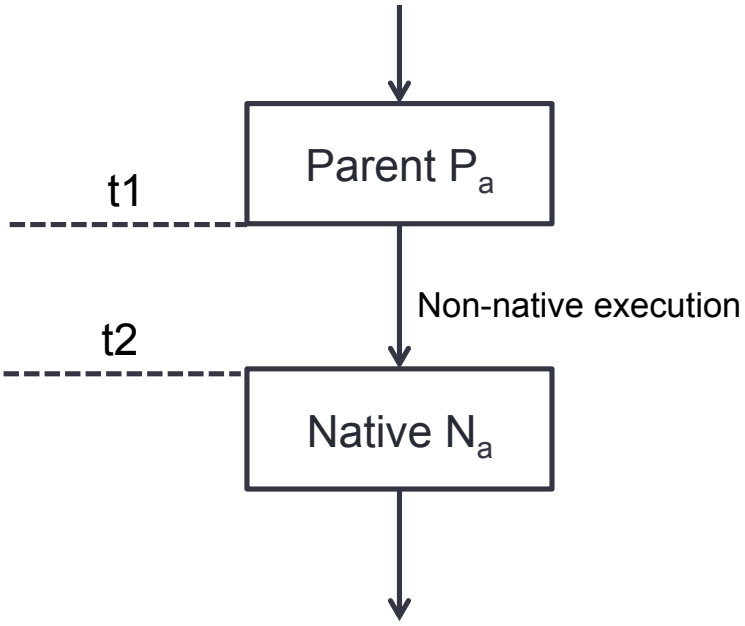

Figure 1: Call Graph with Parent and Native method having non-native execution time $t2-t1$

3.2. Native-to-NextNative Execution Time

The offloading strategy that COMET currently implements is a simple one. It looks at the value of the Round Trip Time (RTT) and the Sync Time to calculate a threshold. The offloading is enabled if this threshold is less than the time elapsed since the last time an abort was invoked at the server. This strategy might not exploit the offloading capabilities to the fullest as it waits for some time before deciding to migrate to the server.
Our analysis on applications revealed that certain applications have a huge chunk of non-native execution code that is computationally intensive after a native function call. Even a simple application that invokes a System.print periodically and has a lot of intense computation between these calls will get affected to some extent by this mechanism. Our static analysis also reveals that significant amount of time is spent in non-native execution between two native calls. In such a scenario, we offload immediately after the first native method returns to the client. With the profiling results, we can isolate the computationally intensive portions of code that are of interest to us and set a flag to inform the scheduler to perform migration without waiting to cross a threshold. The scenario mentioned is shown in Figure 2.

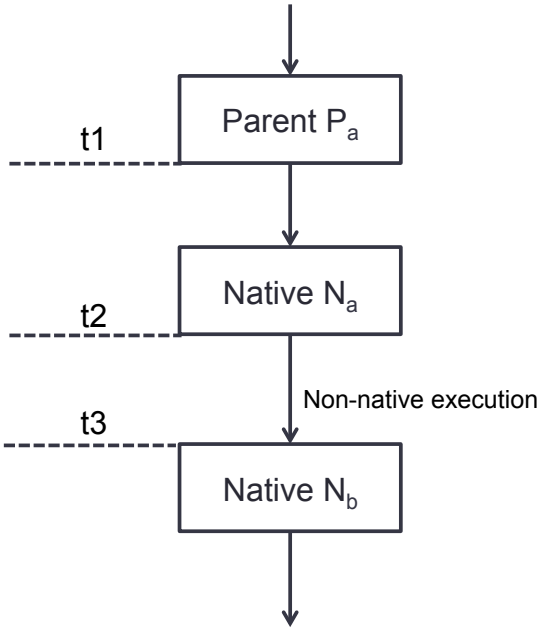

Figure 2: Call Graph with non-native execution time $t3-t2$

3.3. Bandwidth Check

4. Methodology

4.1. Call-Graph Analysis

4.2. Offloading Heuristics

5. Evaluation

5.1. Method Execution Times

5.2. Heuristics Performance

6. Related Work

Recent research has focussed on increasing the computational capabilities of mobile devices. By using code offloading resource intensive mobile components are identified and offloaded to remote servers in order to be executed by machines in the cloud. Most of the relevant work in this area have used code annotations and VM synchronization to implement the client to server offloading. The most noteworthy of these works are MAUI [2] and CloneCloud [1].

CloneCloud follows an approach that is based on static analysis of the mobile application to partition it. Based on this partition, the application can be executed in the client or the server. CloneCloud maintains a clone of the mobile application stack in a virtual machine on the server. This helps is synchronization when the application is offloaded as the application can be executed at the client or the server seamlessly. The strategy proposed by MAUI is based on code annotations to determine the methods that can be offloaded to the cloud. The developer is given the responsibility of generating these annotations in the source code. The MAUI profiler identifies these annotations at run-time and offloads the methods to the server.

Intelligent offloading decisions has been investigated previously by Flores et.al. [3]. Their approach generates offloading decisions based on server parameters. By using fuzzy-logic, they improve the offloading decision over time. This decision engine relies on both, mobile and cloud parameters. Though the ideology of this work is similar to our intention our approach differs in certain aspects. Our approach differs as we look at the execution times of methods, using dynamic profiling to determine when to migrate methods to the server and when to aviod the state transfer and execute locally.

7. Conclusion

References

[1] B.-G. Chun *et al.*, "Clonecloud: Elastic execution between mobile device and cloud," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 301–314. Available: http://doi.acm.org/10.1145/1966445.1966473

[2] E. Cuervo *et al.*, "Maui: Making smartphones last longer with code offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '10. New York, NY, USA: ACM, 2010, pp. 49–62. Available: http://doi.acm.org/10.1145/1814433.1814441

[3] H. Flores and S. Srirama, "Adaptive code offloading for mobile cloud applications: Exploiting fuzzy sets and evidence-based learning," in *Proceeding of the Fourth ACM Workshop on Mobile Cloud Computing and Services*, ser. MCS '13. New York, NY, USA: ACM, 2013, pp. 9–16. Available: http://doi.acm.org.proxy.lib.umich.edu/10.1145/2482981.2482984

[4] M. S. Gordon *et al.*, "Comet: Code offload by migrating execution transparently," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 93–106. Available: http://dl.acm.org/citation.cfm?id=2387880.2387890