Fast Dynamic Execution Offloading for Efficient Mobile Cloud Computing

Seungjun Yang, Yongin Kwon, Yeongpil Cho, Hayoon Yi, Donghyun Kwon, Jonghee Youn, and Yunheung Paek*

Department of Electrical Engineering and Computer Science
Seoul National University
Seoul, Korea
{slyang, yikwon, ypcho, hyyi, dhkwon, youn}@sor.snu.ac.kr, *ypaek@snu.ac.kr

Abstract—In order to meet the increasing demand for high performance in smartphones, recent studies suggested mobile cloud computing techniques that aim to connect the phones to adjacent powerful cloud servers to throw their computational burden to the servers. These techniques often employ execution offloading schemes that migrate a process between machines during its execution. In execution offloading, code regions to be executed on the server are decided statically or dynamically based on the complex analysis on execution time and process state transfer time of every region. Expectedly, the transfer time is a deciding factor for the success of execution offloading. According to our analysis, it is dominated by the total size of heap objects transferred over the network. But previous work did not try hard to minimize this size. Thus in this paper, we introduce novel techniques based on compiler code analysis that effectively reduce the transferred data size by transferring only the essential heap objects. The experiments exhibit that the reduced size positively influences not only the transfer time itself but also the overall effectiveness of execution offloading, and ultimately, improves the performance of our mobile cloud computing significantly in terms of execution time and power consumption.

Keywords—Mobile Computing, Cloud Computing, Code Analysis, Smartphone, Compiler, Execution Offloading

I. INTRODUCTION

Mobile applications are a steadily growing segment of the global software market, whose revenue is expected to reach more than 25 billion dollars by 2015 [1]. As their processing capabilities increase, smartphones are rapidly becoming the computing device of preference that can accommodate most up-to-date mobile applications. Even so, it is still challenging to deliver highly sophisticated applications on smartphones due to the key resource constraints like limited battery, poor processing power and low network bandwidth. Moreover, to minimize device fragmentation [2], virtually all smartphones adopt cross-platform runtime environments, such as Java ME, .NET CF and Android, to develop and run applications. These approaches demand even more resources for storage, computation and energy from a limited supply on smartphones, only to aggravate the constrained resource problem [3].

To alleviate this problem, latest studies suggested various mobile cloud computing techniques that attempt to connect resource-constrained mobile devices to nearby resource-rich powerful clouds [4], [5], [6], [7]. The basic idea is to let devices leverage computation and energy on cloud servers to execute (part of) mobile code that requires heavy use of computing or network resources. People believe that mobile cloud computing opens a new world where smartphones armed with various network connections and rich sensors will extend dramatically its functionalities with help of computational power of clouds. First of all, it seems obvious that mobile cloud computing can accommodate a much wider range of complex applications which have been impractical to run solely on smartphones, such as perception applications, vision, graphics, healthcare, augmented reality and m-learning [7], [8]. As another advantage, mobile cloud computing may relax the design constraints of smartphone hardware which, due to the considerations of size, costs and battery capacity, have been strictly imposed on hardware features like CPU, storage and network [9]. Such strict design constraints often tend to force hardware vendors to put more emphasis on specific features in their products while sacrificing others, depending on their targets in the market. This trend inevitably has resulted in a vast diversity of smartphone products with tremendous gaps in their hardware performance which unfortunately increases the development time and operational costs of mobile applications. The reason is because developers need to provide slightly different versions of an application to cope with variations across a wide range of device capabilities, which may result in multiple executables for a given application [2]. With mobile cloud computing, app developers expect more dependable, consistent performance from their target smartphones backed by computing capabilities of the cloud, as they may let low-end devices rely more heavily on clouds while high-end ones strive in their ways more independently.

To execute mobile code on the remote server such as the cloud or wall-powered PCs, previous work has often employed a technique, called execution offloading, which is the act of transferring execution (or process) between two machines during its run time. By relieving computational loads, the technique labors to bring smartphones benefits in terms of battery and execution time from the servers in their proximity. In execution offloading, there are two key tasks involved before remote execution: code partitioning and state migration. In recent years, there has been a great deal of research conducted to find or support optimal par-
tioning of distributed systems with mobile devices. Some researchers [10] proposed static partitioning schemes where the job assigned to each machine in the system is fixed at compile time. Static partitioning ought to be more doable if the computational resource configurations such as processor speed, memory capacity, energy consumption and network characteristics, remain fairly constant once the process is launched. In mobile computing, however, the configurations can be changed due to user mobility even in the middle of process execution. Therefore, mostly other works [4], [5], [8], [11] have been on the development of dynamic or semi-dynamic partitioning schemes for execution offloading.

In a dynamic partitioning scheme, the code is commonly annotated with directives delimiting the code regions that can, if profitable, be delegated to the server for remote execution. Which regions actually run on the server is decided during code execution when the resource configurations for execution become known. Once a certain region is finally selected at run time, the current state for execution needs to be captured and migrated to the server along with the control command that directs the resumption of the execution. In one approach [5], the entire state including the existing stack and all reachable heap objects is migrated to offload the full process. In the other one [4], the stack is not to be migrated as the functions set to run remotely will be newly invoked in the server. Clearly, there are trade-offs between these two approaches. Above all, the usual amount of state transferred, which is a major decisive factor for the efficacy of execution offloading over mobile networks, is smaller in the latter. In contrast, the former approach relies little on users for code alteration, and supports more versatile code execution because, with the full process in its hand, the server would be able to control the execution more adaptively.

In this paper, we are interested in the full execution offloading approach. Our work is in fact based on the original work of CloneCloud [5], which dynamically offloads part of Android Dalvik code running on a smartphone to the cloud. At the beginning of execution, it analyzes the current resource configurations, and finds a code partition such that the overall execution time or energy consumption on the smartphone is optimized subject to them. In ideal cases where the costs for state migration can be neglected, any code regions except for those using the device resources like GPS and screens would benefit from remote execution. This is obvious because the server processor speed is much faster, and virtually no energy of the mobile device would be consumed while they run on the server. In reality, however, the migration costs may not be neglected but even be a dominant factor that inhibits the regions from executing remotely. The costs are roughly proportional to the size of data to transfer heap objects over the network. As will be detailed in Section II, CloneCloud transfers all reachable objects, whose total size may reach up to multi-megabytes at a time, thus increasing the time for state migration greatly. In our implementation, we have drastically reduced the migration time, which helps us offload more code regions, and so lower the total execution time or energy consumption. This has been achieved by only transferring the essential heap objects, which we define to be the reachable objects that will be possibly accessed within the remotely executing code regions. Such possibility was evaluated by our compiler static analysis techniques. The experimental results seem promising. The migration time is reduced on average by a factor of ten. As a result, our mobile code was able to be offloaded more aggressively at run time, attaining an overall speedup of process execution up to seven.

In Section II, we first explain how CloneCloud works and computes the reachable heap objects for state migration. Then in Section III, we describe our techniques to reduce the amount of migrated state by identifying essential objects from the mobile code. In Section IV, we experimentally demonstrate the effectiveness of our techniques which reduce significantly the execution time of mobile code. Finally, in Section V and Section VI, we related our work with others and conclude.

II. REACHABLE STATE TRANSFER WITH CLONECLOUD

In this section, we discuss how adversely the size of migrated state affects the performance of execution offloading.

A. Overview of CloneCloud

In CloneCloud, a process is an Android phone application running on the Dalvik virtual machine (VM). The process may comprise multiple threads, and some of them, which we call migratable threads, contain the remotely executable methods (REMs) in their code. If a thread has no REM, it will be herein called a resident thread. As a rule, CloneCloud declares a method to be a REM if it does not need to access local resources in the phone such as GPS, cameras and screen. However, the decision on what methods actually will run remotely on the cloud server is deferred until the process starts execution when its computational resources are all revealed. For this decision, CloneCloud implemented two components: profiler and solver.

The profiler measures expected execution times and energy uses of all methods on the phone as well as on the server. It also calculates the average costs required to transfer the state of a migratable thread when the thread is migrated between the machines. The migration may occur at the entry point of any REM, and the state transfer costs change depending on the migration points. Thus the profiler needs to estimate the costs at every possible migration point within a thread. The costs are also influenced by the network characteristics such as loss rate, latency and bandwidth. To reflect these to the results, the profiler takes executions with different network interfaces like 3G or Wi-Fi into consideration during the estimation.

The solver accepts as input all the performance numbers for process executions obtained during profiling, and outputs optimal partitions for the process. A partition is an execution
scenario consisting of a sequence of decisions made at every migration point on whether or not the execution flow and state for the migratable thread must be migrated for or reintegrated from remote execution. Among many candidate partitions for a process, the task of the solver is to choose the best one that minimizes the overall run time and/or energy consumption of the mobile device. Certainly, for different network interfaces through which the machines are connected, the best partitions for the same process may vary. Therefore, the solver generates a collection of partitions each optimized for a unique network condition.

The resulting optimal partitions are stored into the scenario database (DB) shown in Figure 1. When a process is launched, CloneCloud detects the current network status and retrieves the DB to fetch the optimal partition for this process execution. During the execution, if a thread is to be migrated according to the scenario extracted from the DB, the migrator suspends the migrating thread and captures its state. The state is serialized to generate a state package that is then given to the node manager whose mission is to transfer the state package over the network between machines. The state arriving at the cloud is de-serialized and restored into the memory by the migrator on the server. Associated with the transferred state, a new thread is cloned on the cloud and takes over the execution control from the original thread until it must return the control back to the phone according to the scenario. In CloneCloud, the code is assumed to be copied onto the cloud off-line ahead, so at run time no code for the cloned thread is transferred. More details about execution offloading in CloneCloud are referred in their literature [5].

B. Impact of State Package Size on Performance

The state being transferred when a thread is migrated back and forth between machines includes stack, registers, and reachable heap objects (RHOs). Heap objects are composed of class and data objects. A class object is a template describing the behaviors and states that are shared by the objects of its type, and a data object is an instance of a class that contains its local states and methods. RHOs are any heap objects that are accessible or visible in any potential continuing computation. Figure 2 presents an example of heap objects along with two stacks for a face detection process that has two threads: UI and Worker. When the migrator captures the state of a migrating thread, it identifies RHOs by recursively chasing the reference links starting from local data objects in every stack frame of the thread. This procedure is similar to garbage collection. But the migrator looks for live (or reachable) objects, while the garbage collector finds dead ones that have no references to themselves.

In the example, Worker currently has two frames, each of which stores local data objects used by one method in the thread at run time. For instance, the frames for two methods FdView.run and FaceDetector.Facedetection point to the FdView data object and the FaceDetector class object, respectively. Each heap object has reference links to its relevant objects. For example, the FdActivity data object points to its class object and the widget/Button class object. With this data structure, RHOs of each method are determined by checking accessibility of heap objects from its stack frame. Note that almost all heap objects in this example are RHOs because the Fdview data object has multiple links to other class objects as well as the FdActivity data object, which lead to virtually all the other objects in the figure. In Figure 2, the RHOs for the Worker thread are denoted by shaded boxes. Rectangles stand for class objects while rounded ones for data objects. These RHOs will be captured by the migrator for state transfer if Worker is decided to be migrated.

We have discovered that RHOs generally occupy the largest fraction of the state captured by the migrator for transfer. This implies that the node manager would spend most of its time transferring RHOs across the network. Table I shows that the portion of time spent on transporting RHOs during the entire state transfer in two different network environments 3G and Wi-Fi. It confirms our expectation that RHO transfer takes the majority of the state transfer time. This result suggests that minimizing the size of RHOs should reduce the total state transfer time substantially, thereby contributing the reduction of overall execution time of the migratable thread as well.

To explain in more detail the impact of RHO size reduction on the execution time, see the execution scenarios in
Figure 3, demonstrating that different state transfer times can affect the solver’s decision on optimal partitions. We assume here that two methods foo and goo are REMs while gps cannot as it relies on the GPS service on the phone. In Figure 3(a), we list the expected execution time of each method estimated by the profiler. In Figure 3(b), we list the different optimal partitions that might be produced by the solver depending on the amount of state transfer overhead. If the transfer time is 100 ms, the scenario will be chosen as the optimal partition that minimizes the total execution time. Suppose that the time is cut to merely 20 ms by minimizing the total size of RHOs. Even under the same scenario as in Figure 3(b), we can reduce the execution time as depicted in Figure 3(c). In reality, however, the solver would choose the one in Figure 3(d) as the optimal partition since it further accelerates the performance by dispatching the REM goo into the cloud. From this example, we can see that the reduced RHO size will help the solver to find a better partition that exploits more aggressively the computing resources in the cloud, which ultimately can result in dramatic performance improvement in mobile cloud computing.

### III. Essential State Transfer

In this section, we discuss our novel techniques that help us to drastically reduce the size of state transferred at migration points. The central idea behind them is gleaned from the fact that although RHOs are accessible to a thread, not all of them are actually used at run time, and thus that from the transferred state package any object can be removed which has no chance of being accessed during remote execution on the cloud. For this we define a heap object for a thread, called an essential heap object (EHO), to be a RHO that has explicit references in the thread code. Table II compares the size of a RHO set and that of an EHO set for the same thread. From the results in the table, we gleaned the fact that EHOs can be much less in number than RHOs in some applications, as will be exhibited in our experiments where we significantly reduce the state transfer time by not transferring all RHOs, but instead transferring only EHOs.

In the subsections below, we first describe a code analysis technique that is used to extract EHOs from RHOs, and then other techniques that enable us to additionally minimize the time to transfer EHOs.

#### A. Essential Heap Objects

Among RHOs, many class objects come from super classes because Java class objects usually inherit various states and behaviors from their super classes. But in most cases, all the objects defined in the super classes are not required to execute a process; that is, there are some variables or methods never accessed throughout the whole execution. Hence, the cloned thread will safely run on the cloud even if we do not transfer any RHO that will not be referenced by the thread. However, it is almost impossible for us to statically identify which RHO is to be actually referenced at run time. Therefore in our work, we only remove from the transferred state package every super class object (and its related data objects) that has no reference in any method of a migrating thread. By definition above, all the RHOs remaining in the package automatically become the EHOs. According to our experiments, even this conservative approach to isolate EHOs has reduced the state package size to a large extent.

In our work, the unreferenced objects are simply determined by code analysis, where the names of class objects referenced in every method are all stored into a table. When the migrator captures the state, it searches for RHOs by chasing down the relation links in their class hierarchy. When it comes across a class object, it looks up the object in the table. If the object is not found, it is classified as unreferenced. To identify EHOs by finding unreferenced objects, we propose in this work a new component, called the state transfer optimizer, that can be added to the original CloneCloud. Figure 4 shows a new system augmented with the optimizer.

In Figure 2, we showed an example of RHOs for the Worker thread. Here, we also represent the EHOs selected from them with the rectangles or rounded rectangles encased by thick lines. Notice that the FdActivity data object and its relevant class objects are not chosen as EHOs.

---

**Table I**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>RHO transfer (ms)</th>
<th>Total state transfer (ms)</th>
<th>Ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NQ (3G)</td>
<td>49404</td>
<td>51932</td>
<td>95</td>
</tr>
<tr>
<td>NQ (Wifi)</td>
<td>8648</td>
<td>10760</td>
<td>80</td>
</tr>
<tr>
<td>BS (3G)</td>
<td>49596</td>
<td>51542</td>
<td>96</td>
</tr>
<tr>
<td>BS (Wifi)</td>
<td>7166</td>
<td>9362</td>
<td>83</td>
</tr>
</tbody>
</table>

**Table II**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size of a RHOs (KB)</th>
<th>Size of a EHOs (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NQ (n=10)</td>
<td>10807</td>
<td>61</td>
</tr>
<tr>
<td>BS (n=5000)</td>
<td>10899</td>
<td>150</td>
</tr>
</tbody>
</table>

---

(a) Execution time of each method

(b) Migration overhead 100ms

(c) Migration overhead 20ms

(d) Modified scenario for Migration overhead 20ms

**Figure 3.** Impact of state transfer times on the decision of partitions.
We collect all RHos by following Fdview stack frame when the method is invoked. As the case of Fdview.run, this is because the state transfer optimizer reveals through site, only all live. However, if migration occurs at the next invocation of goo, then the address of the second dirty object in a method is identified, the analyzer stores object before corresponding migration point. After every migration point, all EHOs related to this object must be examined for heap objects to be considered dead. When we assemble a state package with EHOs, we only include the live ones obviously because the dead objects will never be used during the rest of execution of a migrated thread even if they are transferred to the cloud.

Depending on migration points in a method, a data object might be live or dead. For instance in Figure 5(b), three data objects local to a method foo are listed, and the ranges of their liveness are also pictured. If goo is a REM, two invocations to goo inside foo would be the migration points. Suppose that their thread is to migrate at the first invocation site following the execution scenario. Then, they are packaged as EHOs and sent to the cloud since they are all live. However, if migration occurs at the next invocation site, only v1 and v2 are live then, (see Figure 6(a)). As a result, the dead object v0 is removed from the original state package as shown in Figure 6(b) and (c). To compute the live range of a data object, we have applied conventional compiler techniques based on def-use analysis [12].

When a local data object is found dead at a migration point, all EHOs related to this object must be examined to determine their liveness (or eligibility of being in the state package). If any of them is also found dead, it will be excluded from the package. As an example, see a data object x of class type B declared inside class A in Figure 5(a). If v0 is not needed in the code, neither is x because x is exclusively accessed within v0. This means that if v0 is dead, so is x. Consequently, as shown in Figure 6, x was deleted from the package along with v0.

C. Dirty/Clean Essential Heap Objects

Once dead EHOs are all filtered out of the state package, the remaining live ones are finally ready to ship. In this last step of state transfer, we have found a way to save the time and energy of transmitting the package over the network. In order to take a glimpse of this idea, see the code in Figure 5(b). Again, let us assume that the thread is about to migrate just before the second invocation of goo. In the code, we can see that v1 has been modified before the migration point while v2 is still intact. The idea here is that it is not necessary to copy and deliver v2 from the phone to the cloud because the exactly same content of v2 can be duplicated simply by creating v2 on the remote site. We call this unmodified object clean and the modified one dirty. In our work, we have used a well-known compiler side-effect analysis technique [12] to identify which objects have been modified before reaching each migration point.

The analyzer accepts application bytecode as its input to identify dirty and clean objects. First, the analyzer seeks every method call (or, a migration point) in a method when it explores its input code. For each local data object in a method, it labels the object and its relevant objects as dirty is there is any instruction which assigns any value to the object before corresponding migration point. After every dirty object in a method is identified, the analyzer stores the ID of each dirty object into a table with the address of the object before corresponding migration point. For instance in Figure 5(b), the ID of the dirty object v2 and the address of the second invocation of goo are stored to the table. Notice that relevant heap objects to v2 are not stored.

As stated in Section III-B, live EHOs are only included to the state package in our work. When the migrator chases every stack frame and packs each live EHO in a frame to the package, it also confirms whether the object is dirty or not by searching for the object in the table. If the ID of the object is not in the table, then the object is clean. In this case, the migrator creates a stub for the clean EHO and adds it to the state package, instead of adding the EHO.
itself and its relevant objects. A stub contains information of an object such as ID, class name, and the address of an object necessary for the migrator on the cloud to create new instance of clean EHO. Comparing to the size of clean EHO and its relevant objects, the size of a stub is much smaller, being only a few bytes. For the reason, the state package size can be reduced by substituting the stubs for clean EHOs and their relevant objects. From stubs in the reduced state package, the migrator on the cloud side creates new instances of the clean EHOs and links them to the transferred dirty EHOs. We named this on-the-fly instantiation of the clean EHOs on-site-duplication.

Figure 7 shows state restoration via state copy and our on-site duplication, after the migration occurs at the second invocation of `goo` in Figure 5(b). Because `v2` is clean in this case, the reduced state package in Figure 7(a) does not include `v2` and its relevant object `x`; instead, the stub for `v2` is included in the package. The on-site duplicator in Figure 7(b) uses this stub to call the constructor method of `v2`. After this on-site duplication, new instances of `v2` and `x` are created. Then, the migrator restores the complete state for remote execution by assembling them together with the dirty objects `v1` and its relevant objects which are just copied from the phone through the state package.

**IV. EVALUATION**

We implemented our execution offloading model including the state transfer optimizer and the modified migrator on the Android 4.0.3 branch, and tested it on the smartphone and the server, respectively. The smartphone is a Galaxy Nexus with dual-core 1.2 GHz CPU and 1 GB of RAM. For the server, we used a quad-core desktop with a 3.1 GHz CPU and 8 GB of RAM running Ubuntu 11.10. To execute Android on a regular Intel x86 desktop, we built a target of Android for VirtualBox 4.1.8. We also used an off-board equipment [13] to profile power consumption of the smartphone.

To evaluate the effectiveness our offloading model, we implemented three benchmark applications in different categories: CPU, IO, and user interactive. In the following subsections, we describe our applications and its experimental results in detail.

**A. CPU and IO benchmark**

As our CPU tasks, we chose three simple kernels, N-Queens problem(NQ), BubbleSort(BS![](image)), and a FIBONACCI sequence generator. For a given input n, the NQ problem finds all possible ways to place n queens on a chessboard, whose size is n by n, so that no queen would attack any other queen. The BS is a well-known sorting problem. In Tables I and II, the experimental results for these two codes were already presented. The FIBONACCI sequence generator recursively calls its member method to generate a FIBONACCI sequence for a given size of a sequence. Because the performance of this application depends mainly on the CPU power of the device, we classified it as a CPU task. We also implemented the face detector, which recognizes all faces in a given image. After all faces are found, the detector draws green rectangles on the each face. We used OpenCV 2.3.1 library for Android to implement our detector. The detector downloads input images from an external on-line server at run time. We chose it as a IO task, since the computation heavily relies on IOs involving network operations and file accesses. For the evaluation, we vary the size of the sequence between 25 and 42 for the generator, and the number of images from 1 to 99 for the detector. To profile our applications, we used a set of randomly generated inputs. By using the profiling result, we solved the partitioning problem in a similar way to CloneCloud [5], and built the partitioning scenario for each application. We also applied our static analysis based on the compiler technique to reduce the state transfer time, which is mentioned in Section III.

Figure 8 shows execution times and power consumption of the smartphone for three applications on the largest input size. In Figures 8(b) and (d), the measurement for phone-
alone execution is divided to different bars ("Phone.3G" and "Phone.Wi-Fi") because our detector downloads its input images from an external on-line server at run time, therefore the network latency affects its execution.

For the largest input size of the face detector, we obtained a speedup of 6.6 on the smartphone over 3G, and 5.7 over Wi-Fi. Our approach achieved much higher improvement than the RHO approach, whose speedup is 2.1 over 3G and 3.7 over Wi-Fi. It is induced by reducing the state transfer time from 47 second to 3.7 second over 3G, and from 9.4 second to 0.6 second over Wi-Fi. We also improved the performance of the RHO approach by about 65% over 3G and 11% over Wi-Fi. Notice that our approach is more effective over 3G than Wi-Fi. We believe that such a result is caused by different network latency: due to the greater latency and lower bandwidth, the migration cost over 3G network is much higher than Wi-Fi. Similar to CloneCloud’s result [5], power consumption generally follows execution time.

Table III shows the evaluation result for the other input size of two applications. The RHO approach failed to offload the FIBONACCI generator and the face detector over 3G. In contrast, our approach succeeded to offload both of them. This result demonstrates that the reduction of the state transfer time really has great impacts on the performance of execution offloading, as predicted earlier.

B. User interactive chess game

Another benchmark that we tested is a chess engine which is a central part of a user interactive chess program. The engine accepts the user’s move as the input, and returns a ‘counter move’ given the position of each chess piece on the board. To find the optimal counter move, it uses a simple minimax algorithm: it considers all possible next move and scores them by traversing a game tree, which is a directed graph whose nodes are positions of each chess piece and whose edges are moves. After that, it returns the highest scored move. If there are more than one moves whose scores are equal, the engine chooses one of them randomly.

Our chess engine invokes a key REM, `getNextMove`, to obtain the optimal counter move. As stated in Section II, the solver makes an offloading decision for the REM based on the profiling result which is, in the original implementation [5], the average of execution time and power consumption of an application on a set of randomly generated inputs. Such a strategy might be acceptable for some applications whose execution times are relatively consistent regardless of the sequence of their input. According to our analysis, however, this simple strategy is not workable for others like our chess engine whose execution time drastically varies on its input values. To explain this, see from Figure 10 the execution times of our engine on the phone which are measured and plotted every time a user input is given. One noticeable thing here is the similarity between two curves of these time plots: the execution times rapidly hike as the matches start, but after reaching the top at the early stages, they both gradually drop as the matches come close to an end. Among various factors resulting in this execution pattern, a major one we found is the number of pieces left on the board. According to our analysis, however, this simple strategy is not workable for others like our chess engine whose execution time drastically varies on its input values. To explain this, see from Figure 10 the execution times of our engine on the phone which are measured and plotted every time a user input is given. One noticeable thing here is the similarity between two curves of these time plots: the execution times rapidly hike as the matches start, but after reaching the top at the early stages, they both gradually drop as the matches come close to an end. Among various factors resulting in this execution pattern, a major one we found is the number of pieces left on the board. Among various factors resulting in this execution pattern, a major one we found is the number of pieces left on the board.

Figure 8. Average phone execution times and power consumptions for FIBONACCI sequence generator and face detector application with the largest input size. The overhead for the state transfer is also annotated.

Figure 9. Average execution times of `getNextMove` for each distinct number of pieces left on the board.

Table III

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>RHO.3G(s)</th>
<th>RHO.Wi-Fi(s)</th>
<th>EHO.3G(s)</th>
<th>EHO.Wi-Fi(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIBONACCI(n=34)</td>
<td>N/A</td>
<td>N/A</td>
<td>3.215</td>
<td>2.023</td>
</tr>
</tbody>
</table>

Figure 10. Average run times of `getNextMove` for each distinct number of live pieces.
transfer) respectively. Therefore in the original design, the solver may decide that offloading the REM over 3G is always profitable. This naive decision, however, will result in the performance loss for some cases like those with less than 10 remaining pieces where running the REM on the phone is clearly more profitable as shown in the Figure 9. Consequently in our new design, the profiler estimates the execution times of the chess engine REMs for each different number of pieces on the board, and the solver makes variable offloading decisions for the same REM depending on the number of pieces. In this experiment, we have 31 decision points for \texttt{getNextMove} over 3G and Wi-Fi, respectively. For each point, the migrator either offloads the REM or not at run time, following the decision produced by the solver.

In Figure 10, the performance results of our execution offloading strategy for the chess engine are presented. We played each game until 2 pieces were left. Each play consists of 68 and 142 moves, respectively. For the play with 68 moves, we obtained a speedup of 2.7 over 3G and 5.4 over Wi-Fi and for the play with 142 moves, 2.2 over 3G and 4.5 over Wi-Fi. Even if our variable decision strategy was sometimes incorrect, thereby causing occasional performance loss as in the cases of the 41st move in Figure 10(a) and the 45th move in Figure 10(b), it was proven to be correct for most cases, achieving overall significant performance gains from execution offloading. The power consumption of the chess engine on two plays is also shown in Table IV, demonstrating the effectiveness of our strategy in terms of power saving.

\begin{table}
\centering
\begin{tabular}{|l|c|c|c|}
\hline
Play & Phone(A*h) & EHO.3G(A*h) & EHO.Wi-Fi(A*h) \\
\hline
68 moves & 122.39 & 107.98 & 28.44 \\
142 moves & 165.05 & 169.29 & 71.72 \\
\hline
\end{tabular}
\caption{Average phone execution times of the chess engine on two plays. Time unit is second.}
\end{table}

V. RELATED WORK

One of the earliest studies that aim to empower portable devices with surrounding servers was done by Satyarnarayanan et al. [9], who have developed versions of \textit{ISR} systems for the past decade. To offload a process running on the device, they migrated the full VM or OS image along with the process. Not surprisingly, the amount of transferred data for \textit{VM migration} tends to be huge (around in the order of gigabytes). To lighten the load, they proposed the \textit{dynamic VM synthesis} approach [7] where a small VM overlay is sent by a mobile device to the \textit{cloudlet} (nearby small cloud) that is already installed with the base VM which the overlay was derived from. The overlay size was reported about one order of magnitude smaller than the full VM size, so they claimed that the approach might be feasible for mobile computing using fast wireless LANs like Wi-Fi. However, even that figure would be still too high for lower bandwidth WAN interfaces like 3G.

In order to make mobile cloud computing more viable over the wireless WANs, many recent studies listed below have proposed \textit{process-level migration} approaches that normally require only a few megabytes [4] for each state migration. As classified in Section I, these approaches can be divided largely into two groups: those using static partitioning schemes and those using dynamic ones. A noticeable work in the first group might be Wishbone [10], which gives a solution for optimal partitioning of sensor network application code across sensors and servers. It statically partitions the application code based on profile data that include the computational and network load by using an \textit{integer linear program} to find the minimum use of CPU and network bandwidth. Wishbone guarantees that the optimal partitioning can be predetermined regardless of the target hardware platform because it targets a confined area of applications where a division of subtasks is fairly clear.

In static partitioning schemes, using the programming models provided by the middleware or APIs, the users must manually specify at compile time how to partition their code, what state to migrate, and how to adjust the offloading strategy to the varying network status. To relieve users from such burdens, a majority of studies have been interested in
dynamic or semi-dynamic partitioning schemes. One of the first ones is OLIE [3], which collects the current status of the memory utilization and available network bandwidth to decide whether offloading should be triggered at run time. But the main goal of OLIE is to overcome only the memory resource constraints of mobile devices. This is deemed relatively simple as compared to optimizing energy consumption and execution times, which is our goal like others [4], [5]. As another example, Odessa [8] dynamically partitions applications using a greedy algorithm, and adaptively makes offloading decisions. However, the developer who tries to apply their approach must use the specific development framework. Giurgiu et al. [11] propose an elaborate system that dynamically distributes several components of an application between a server and a smartphone. The system is realized on top of their middleware that can support the actual distributed deployment of an application between machines. However, the application must be coded in a special language in order to be worked with this approach, while we support ordinary Java code.

CloneCloud [5] suggests dynamic execution offloading approach by modifying the mobile execution environment, Dalvik VM, to capture the current execution state. CloneCloud can reduce the run time overhead, because they do not need to modify the application code while some approaches have to do. Some approach appends new statements to the applications code to do that, because they do not want to modify the execution environment to keep the flexibility. Due to the new appended code, a significant overhead may be incurred on applications performance [6]. There are some approaches such as MAUI [4] that labor to reduce the runtime overhead. MAUI is a RPC based offloading architecture which decides at run-time which methods should be remotely executed based on the best energy savings possible under the mobile devices current connectivity constrains. MAUI requests user annotation on the application code to mark migratable methods. Ma et al. [6] suggests a Java bytecode transformation technique to migrate computation from a mobile device to a server based on Java exception handling mechanism without imposing significant overhead on normal execution. But it still has much overhead when migration is taken place.

None of them did not explicitly discuss how to use compiler static analysis to reduce the amount of migrated state.

VI. CONCLUSION

In this paper, we discussed our recent research of assisting execution offloading by reducing the size of transferred state. While previous work based on the full execution offloading has focused on finding optimal partitions for given computational resources and network conditions, they did not make active effort to reduce the state size which, as we proved, has been a crucial element for the success of execution offloading. In this paper, we have demonstrated that careful compiler analysis greatly helped our optimization techniques to effectively achieve our research goal, thereby enhancing the efficiency of mobile computing with the computational support of clouds.

ACKNOWLEDGMENT

This work was supported by the Korea Science and Engineering Foundation (KOSEF) NRL Program grant (No. 2008-0060166), the Engineering Research Center of Excellence Program (Grant 2012-0000470), the Center for Integrated Smart Sensors as Global Frontier Project(CISSL20126054193) funded by the Korea government (MEST), and the Business for Cooperative R&D between Industry, Academy, and Research Institute(Grant C0019562) funded by Korea Small and Medium Business Administration in 2012.

REFERENCES