Bypassing Android Hardware Limitations

Offloading Model

Carleton University, COMP 4905
Winter 2017
Prepared By: Kerolos Gattas
Supervisor: Dr. Dwight Deugo

Table of Contents

1. INTRODUCTION

Offloading has been a topic that researchers are utilizing to increase the power and to bypass the limitations of device's hardware. One of the popular ones is data offloading that helps users achieve better connectivity with their devices. Another offloading area that is currently being researched is the ability of the device to send a heavy task to be executed on the cloud or on a server where it can be executed much faster and much more efficiently. This paper will examine task offloading where android phones will send heavy tasks to a server that is running on a PC with much better hardware than those that are usually found on android devices. This paper will also examine the possibility of a generic offloading engine that developers can use when developing their android applications that lets them bypass the hardware limitations.

1.1 Problem

The usage of smartphones today is higher than ever before and it keeps increasing each year [1]. Some people depend almost entirely on their smartphones for all their technology needs. This means that they either can not perform heavy software tasks in a quick way or that they would have to use other devices that may not be as portable as smartphones or that they may not own. Smartphones have a lot of capabilities and functions that allows diverse users to be able to utilize it in many ways.

Android is considered one of the biggest smartphone operating systems; their ability to target a diverse audience allows them to expand and utilize new technologies. Android's play store allows users to find the applications that accomplishes their tasks and that help them in their everyday lives. Thousands of applications exist on the play store as well as games, books, movies and music. The play store allows independent and big developers to upload their applications where users can download or buy their content. This allows for a more diverse application base where users can find what they desire built by developers that are expert in their field.

Android developers however face several problems in their development stages especially concerning optimization. The first major problem is the hardware limitations of smartphones. Smartphone's hardware is usually weak compared to other devices like a home pc. This limits the software capabilities of the device compared to today's technology since the hardware can not handle heavy tasks. The second major problem which exists especially with android phones is the number of different devices and different hardware specifications for each device. This means that developers have to account for all these different devices and differnet hardware to be able to target the most amount of devices they can with their products. This makes the development of android applications very limited as well as very inefficient since optimization is a huge step in development. This also makes the developers compromise quality for efficiency to develop a marketable application that would target most devices.

A solution for the hardware limitations is to use the software powers to solve it. Task offloading is a solution for this problem that allows smartphones to perform heavy tasks efficiently and fast without the need for a powerful hardware on the device.

A problem with offloading is that it depends on the connectivity of smartphones, which means that it is a much more powerful solution when using WI-FI instead of the mobile's data network which limits the offloading solution. However, several offloading models try to solve that problem by using efficient methods to transfer the data across the network. Other ones try to calculate the offloading cost and compare it to performing the task on the device itself and then deciding on whether it is better to offload the task or not according to their current connection [2].

1.2 Motivation

Working on the offloading problem before made me realize that most of the solutions are focused around making the models as efficient as possible. However, none of the models that I have read focus on providing a model that developers can use. I believe that once an efficient model is produced, then we will see a product that developers can use. However, since we don't have a definite efficient model yet, I wanted to try to develop my own model that developers can use as well as compare the results of offloading tasks vs not offloading.

Another motivation is that as a developer and a gamer myself, I tend to compare the quality of games in terms of their user interface, controls and their graphics on smartphones to ones that are on PC and consoles. Right away one can see the

limitations of the smartphones in the games category. Games on smartphones suffer from optimization problems more than the ones found on PC or consoles, a lot of games usually run very well on the high-end smartphones but they perform poorly on other smartphones with lower hardware specifications. Reaching the level of offloading that will allow us some day to be able to play the same type and quality of games on our smartphones that are found on PC and consoles is something that I want to help develop some day.

This project also allows me to create and experiment with the offloading topic on my own. It helps me develop a more refined model in the future, as well as it could be a research topic that I can develop when doing my graduate degree in the future. This is definitely a topic that I think will be a big part of our future as it allows us to be limitless in our device's hardware.

1.3 Goals

This project has two goals: the first is to show and compare the performance of the offloading model vs executing the tasks on an android device. The second is to provide a generic offloading engine that other developers can use that only requires them to send method parameters to the server and then receive the results. Both goals and their results will be discussed in more detail in section 4.

The first goal is showing how offloading improves the performance of applications on android when they have heavy tasks. In this paper, different graphs will be presented that compare the running time of heavy tasks running on an android

device vs if they were offloaded. The following is going to be compared in the results section using graphs and analysis: generic offloading vs direct offloading (where direct offloading means that the server has a specific method that handles the offloaded method vs a generic one where the server has a generic method that handles any offloaded method) and offloading vs executing the task on the device.

In the second goal, the generic offloading engine will be discussed as well as its requirements. The plan was to have a very simple offloading engine where other developers can use without having to write more code. However, during the development stages, building the engine in such a way was not possible. Instead, there are two areas where the developers will have to write code for the engine or alternatively, they can build their own offloading method on the server where a sample example will be found. This will be discussed in greater detail later.

1.4 Objectives

The main purpose of this project is to evaluate the speedup factor that the offloading engine provides as well as examining the possibility of creating a generic offloading engine that developers can use without the need of writing a lot of code. This project aims to see how complicated building an offloading model is and to explore the different ways that it can be implemented.

Other objectives for this project is to explore and build android applications that contain heavy tasks in regards of their execution. These applications usually require powerful hardware's to run efficiently. The two applications that were developed run

efficiently with small values, but with larger values they have much larger execution time.

1.5 Outline

Section 2 will examine the project's background and introduce the project at a higher level. Section 3 discusses the project's approach and implementation details. Section 4 will reveal the results and the effectiveness of the offloading model. Finally, section 5 will conclude the paper and introduce opportunities for future work and improvements for the generic offloading engine.

2. BACKGROUND

2.1 Generic Offloading VS. Direct Offloading

In this project, two methods for task offloading will be introduced. The first is a generic one that has several requirements for the offloaded methods. The second is a direct offloading method where the offloaded method is being called directly from a specific POST method. Both methods produce close results but as the values of the parameters of the offloaded method becomes bigger, one can start to notice a difference in performance between the two.

The difference in performance between the two methods is mainly caused by reflection in the generic model. Reflection is a Java API that is used to examine and modify Java code during runtime/execution. Reflection offers a lot of tools that allows the developer to examine classes, variables, methods, constructors and even annotations. One example of reflection is that JUnit uses it to find methods with test annotations so that it could invoke them [3]. Another example is in an IDE like eclipse when it auto completes a method or a variables name [3].

Reflection, however, has a lot of disadvantages. The first issue is specific to the project; reflection is dependent on the developer sending the right method name to the server when using the generic model. Human errors are possible, with reflection one loses compile-time type safety which means that the server can run into a runtime error if it receives the wrong method name.

Another major issue with reflection is that it is a very slow process. Reflection usually scans classes and methods which usually consumes a lot of time [4]. This means that when the generic offloading model is trying to find the desired method it will consume more time to find that specific method using reflection rather than calling it directly at compile time.

The last disadvantage of reflection that concerns this project is that it requires the program to know the exact name and parameters of the offloaded method during runtime. This means that the developer must insure that they send the correct data for the generic model to use it, otherwise they might receive an error message from the server. Since the parameters are being sent from the client to the server that limits the object type that the parameters can use. Thus, the parameters for the offloaded method must be of type Object. Also since the client is sending the parameters to the server and the server is then trying to find the offloaded method; a security issue might arise and malicious content could be in the parameters but this will not be examined in this paper as security is not the focus for this project.

All the above-mentioned disadvantages make the generic engine much worse and much more dangerous and although, the generic engine requires less code from the developers to write, it is a much slower method to offload the tasks to the server and can cause a lot of errors.

For direct offloading, the runtime of the offloading is much better than the generic one with higher values for the offloaded methods. However, direct offloading

requires that the developer writes the direct offloading code on the server to achieve better offloading results. This might present a problem for developers that are unfamiliar with the server's code and developers that may have not worked with the server's environment before. Thus, both models have issues but if the overhead performance of the generic one is something that can be improved or even ignored, then the generic model is a much better solution for the developers.

2.2 Related Work

Several recent research papers have been examining the idea of task offloading to either a server or the cloud. Several models have been developed with different focuses on different areas for this problem. In this subsection, several papers will be presented and analyzed in their designs and their solutions.

The first paper presents a statistical cost model to calculate the cost of offloading rather than calculating the average cost as in other models. The offloading model uses an Execution Dependency Tree (EDT) to calculate the offloading cost of each module and then later calculating the average for the entire application using the EDT. The model calculates the cost and decides at runtime if it is better to offload the code or not [5].

Another model called "Making Smartphones Last Longer With Code Offload (MAUI)" uses its offloading model to conserve energy power (battery) of the smartphone. MAUI creates two versions of a smartphone application, one of them would run on the smartphone locally while the other would run on the server. It then

uses reflection and sterilization to determine the networking, the energy consumption and the CPU costs; using these costs the application is partitioned at runtime where parts of it are offloaded to the server and the rest is executed locally on the smartphone. MAUI also does not require the developer to write extra code to make the application compactable with the offloading engine, it only requires that the developers mark the method that could be offloaded with the “removable” annotations in their code [2].

The final model “ThinkAir” uses virtual machines on the cloud to perform task offloading. ThinkAir takes advantage of parallel computing to perform the offloading task on the cloud where VMs are dynamically created, destroyed and resumed to handle the offloading. This model allows the offloading to be efficient and powerful since several VMs are used, which helps in exceeding the limitations of the smartphones [6].

All these applications determine whether to offload the code or not at runtime. Each one of them solve the offloading problem in a different way, however, all three produce positive results. It is hard to compare all three models and report which one is the best since they are different in their solutions as well as their focuses might be different (MAUI focuses on conserving energy). None of them offers a definite generic model, MAUI offers one that might be the closest to a generic solution but the model also focuses on conserving energy more than exceeding the hardware limitation of a smartphone. All three models are advanced models when compared to the model for this project. If one of these models can be integrated into the generic model from this

project then one can start exploring the idea of task offloading in a way where smartphone developers can use these models, since all of them are more research models than they are for the developers.

3. APPROACH

3.1 Offloading Server Design

The server for this project is a Maven Web Application that runs on a Tomcat server, that uses Jersey to develop a RESTful Web Service. The server requires Jersey dependencies and the JSON-SIMPLE dependency to work properly. All the offloading methods use HTTP POST Methods.

3.1.1Direct Offloading

The direct offloading means that the developer of the application has to write a POST method that would communicate with the client's android device and that would handle the offloading. There are two steps when using the direct offloading model. The first step is to determine which methods should be offloaded in the application and then creating a separate class on the server that will contain the offloaded methods. The second step is to write the POST method that will handle the offloading. The POST method should receive the parameters for the offloaded method, then calls the proper offloaded method and sends the result back to the android device. The design options are open to the developers, whether they would want to send and receive Strings or if they would want to use JSON Objects for their data transfers.

The N-Queens direct offloading method receives the parameter n as a string that represents the number of queens and the board size. The method

sends the solution of the puzzle as a JSON Object that contains a JSON array. The N-Queens method takes in two parameters, the first is an empty array of size n and the second one is an integer n. Since the array is an empty one, then the client only sends the integer n parameter, and the server can then create a new array and initialize it for the first parameter of the offloaded method. This an example of direct offloading where not all the parameters needed to be sent to the server, since the server knows which method will be offloaded at compile time and can manage the method much better than a generic one would.

The direct offloading for the three Math algorithms in the Math application are the same so rather than discussing each separately, all three will be analyzed together. The nth prime and the Fibonacci algorithms take an integer n as the method's parameter while the factorials algorithm takes a BigInteger as the method's parameter. All three POST methods receive the parameter as a String and then convert it to the appropriate type when calling the offloaded method. The result for all three methods is sent as a String even though factorials and Fibonacci return a BigInteger and nth prime returns an integer. These three methods are simple example of how all the parameters are sent and how the server and the client are only sending and receiving Strings. The three POST methods and the N-Queens POST method provide two different examples that developers can refer to when using direct offloading methods so that they can build their own POST methods.

3.1.2Generic Offloading

The generic offloading model makes it easier for developers to use offloading for their applications, however, it comes with a cost associated with performance and runtime. The model uses reflection to handle the offloading of any method, however as discussed above, reflection has a lot of disadvantages that might make it worse. If these disadvantages like the runtime difference between the generic and the direct offloading models do not matter to the developer then the generic offloading model makes it much easier for the developer since they do not have to write their own offloading POST method.

In its current form, the generic model has several restrictions and requirements that must be met to offload the code properly:

1- The offloaded method must be static.

2- All parameters/arguments must be of type Object.

3- The return type must be an Object.

The method must be static because it makes it easier for reflection when invoking the method; the generic model does not have to specify an object to invoke the method on. All parameters/arguments must be of type Object since the reflection method does not know the type of arguments, then the type must be unified for the offloaded methods. Finally, the return type must be an Object is to unify all the return types. This makes the conversion of the return type be handled by the developer on both the server and the client side. The generic

method will send and receive Objects, the conversions must be handled by the client and the offloaded method. This design makes the generic engine less vulnerable to type cast/conversion errors during runtime.

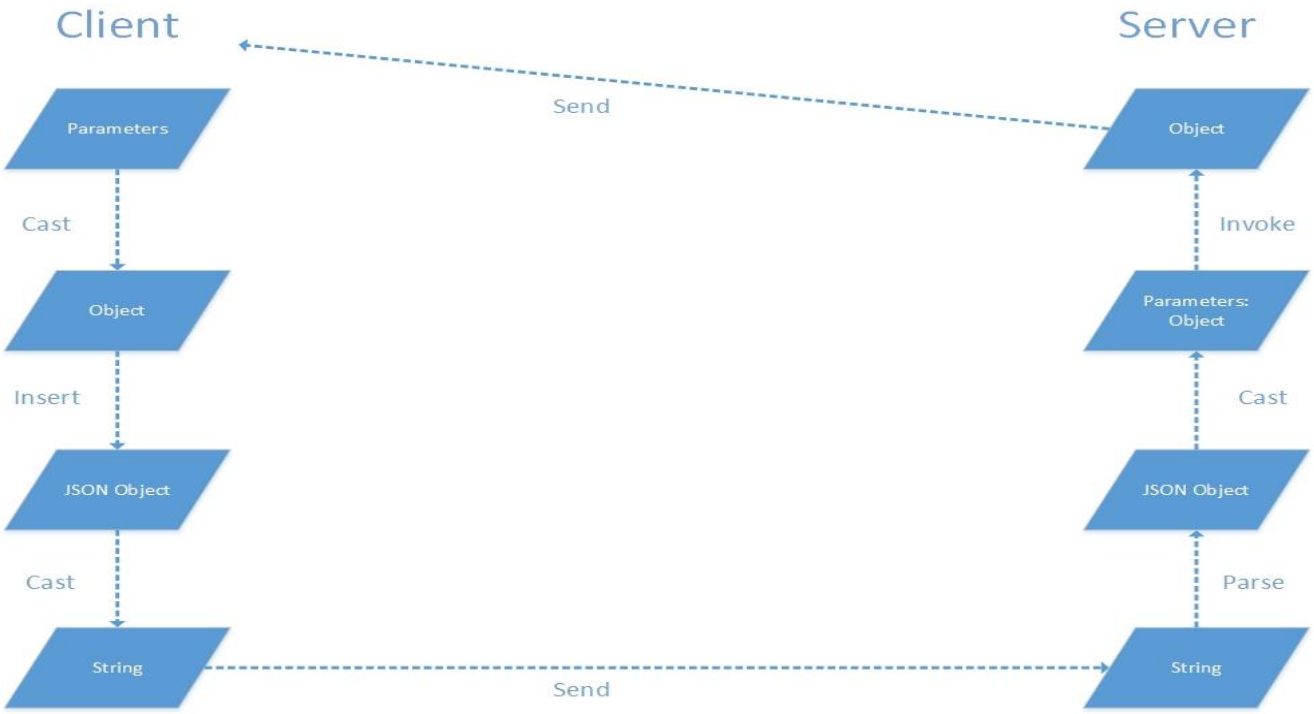


Fig. 1 Generic Offloading Communication Steps

To use the generic offloading model, several steps must be followed. Figure 1 illustrates the steps that the generic model requires. First, the client must cast all the parameters/arguments for the offloaded method to an Object type. After that the client will insert the parameters into a JSON array that will be inserted into a JSON Object. Finally the client calls the toString() method on the JSON Object and sends it to the server along with the name of the offloaded method. When the server receives the String that contains the parameters, it

first finds the offloaded method using reflection. It acquires all the offloaded methods and then searches for the specified offloaded method by the name that the client has sent. It uses a JSON Parser to cast the string back to a JSON Object. After that the server creates an Object array and inserts all the parameters in that array. It then invokes the offloaded method with the Object array as the parameter and stores the result in an Object variable which it then sends to the client.

The generic model requires work from the client side that only affects their code, thus, eliminating the need for the developer to understand the server code to create their own offloading POST method. However, since there are several type castings, complicated parameters like class objects and arrays might not work and that was the case with the N-Queens method. The N-Queens method's first parameter is an array, using the above model would cause a runtime exception of illegal casting when trying to offload the N-Queens with the generic model.

3.2 Android Applications Design (Clients)

3.2.1N-Queens Puzzle

N-Queens puzzle is a famous problem where the algorithm is supposed to place a specific number of queens on a n*n chess board such that no two queens can attack each other. Therefore, the solution must be that no two queens exist

on the same row, column or diagonal. N-Queens solution can be found for all n except n=2 and n=3. With large values of n, the algorithm would need a lot of resources from the device to find all the solutions for n; thus, this makes the N-Queens puzzle a perfect game to test task offloading.

The code that solves the N-Queens puzzle is not my own [7] but we will still explore how it works and how it finds all the solutions to the problem. The application interface is a simple one where the user would enter an integer that represents n and the application will then solve the puzzle locally on the smartphone as well as on the server using direct offloading. The application will display the solution for the puzzle and the runtime for both the offloading and local executions.

The code that solves the puzzle loops through all the queens. For each queen, it loops through the chess board placing the queen during the loop, in each iteration the code checks if the location is valid or not by checking if the placed queen threatens any other queen. If the queen does not threaten any other queen, then it places it on the board and starts to find a location for the next queen. When the algorithm has placed n queens it adds the solution to an array list and terminates.

The application sends the parameter n and receives the solution from the offloaded method using Volley. Volley is an HTTP library that can be used by android devices to make networking easier and faster. The application uses a

custom request class to receive the solution for the puzzle in a JSON Object. The application only displays the solution that is found locally, since some of the solutions could be quite large and displaying them twice would be an overhead. Finally, the application keeps track of the execution runtime of the offloading method and the one that is executed locally and prints those runtimes on the android device along with the puzzle solution.

3.2.2Math Application

Several Math formulas and applications require a lot of steps, especially ones that require iteration or recursion. These applications require a lot of hardware resources when the input for the problem is big. In the Math application, three Math algorithms were developed that require a lot of resources. The three algorithms are: calculate the nth prime number, calculate the factorial of a non-negative integer and calculate the nth Fibonacci number. The interface for the application is pretty much the same as the one for the N-Queens application. The only difference is that there is a menu at the beginning that lets the user choose which math algorithm they would like to execute. Similar to the N-Queens application, the user will enter an integer input for any of the three algorithms and will then get the solution and the runtime of that algorithm. The math application offloads the method using generic and direct offloading.

A prime number can only be divided with no reminder by itself and by 1. Finding the nth prime numbers is quite an easy process but one that can be long when n is large. To find the nth prime number we loop through all the numbers starting from 2, for each number we test if it is a prime number or not, if it is then we increase our counter, we find the nth prime number once the counter is equal to n. Since for each number we need to check if it is a prime number or not, that means that a loop is created where the desired number is divided by each number that comes before it, if there is a reminder in all of these operations then the number is prime. This means that we essentially have two loops, the outer one loops through all the numbers starting at 2 and the inner one loops through all the numbers that come before our desired number.

The factorial for a non-negative integer is the product of all positive integers that are less than or equal to n where n is the input number to the factorial problem denoted by n! To Find the factorial of n, we loop through all the numbers that are less than or equal to n and multiply them together. Factorials unlike the nth prime number require a lot of resources since they grow faster. Factorials eventually grow faster than an exponential expression. This means that huge numbers are being multiplied by each other which requires a lot of CPU power to acquire the solution.

The Fibonacci numbers are a sequence of numbers where each number is the result of the sum of the two preceding numbers. It is defined as:

$F_n = F_{n-1} + F_{n-2}$. Fibonacci algorithms are usually calculated either by using recursion or iteration. Recursion is used when there are no big inputs, but since this project measures performance and runtime; iteration is used to solve the problem. The algorithm creates an array that stores the value of all the Fibonacci numbers that are calculated so far. In each iteration, the Fibonacci value is calculated and the second to last Fibonacci is stored in the array. To calculate the nth Fibonacci where n is a large value, requires a lot of loops which is a heavy task that requires a lot of CPU power and resources.

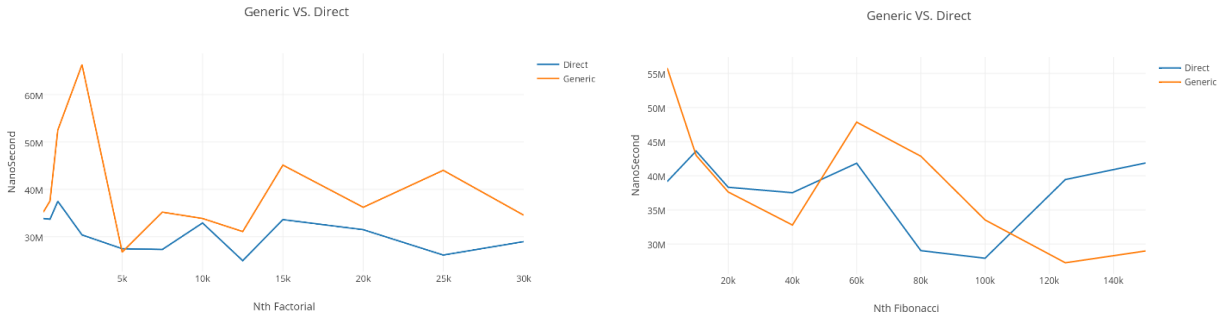
All three algorithms require powerful hardware resources for large inputs. Thus, the algorithms are perfect to test task offloading. The algorithms for the Math application use both direct and generic offloading. Volley is used to send the parameters to the server and to receive the solution to the problem. For the direct offloading, the algorithms receive the solution as a string that they can convert to the appropriate type. For the generic offloading, the parameter must be converted to an Object type that is inserted into JSON Object and then the algorithm sends the JSON Object as a string to the server. The result is received in the form of a JSON Object from the server that contains a JSON array with the results. All these conversions are required for the generic offloading model. All three algorithms keep track of the runtime of the local, direct offloading and generic offloading executions. The application displays the results of the algorithm that is executed locally since the results can be quite large. The runtime for all three methods is printed to the screen device.

4. RESULTS/VALIDATION

Offloading improves the execution time with large inputs for the Math application and the N-Queens puzzle. The main goal of this section is to show the speedup factor that the offloading achieved. Although offloading is worse with smaller values it improves the runtime for large inputs dramatically. This is one of the reasons why some offloading models calculate the cost at runtime so that if its not beneficial to offload the model, then it would not offload the method and it would execute it locally instead. This section will examine and analyze generic offloading, direct offloading and local execution.

Before analyzing the result, device specifications will be listed to see how the hardware affects the performance of offloading. The specifications for the PC that is running the server: 12.0 GB RAM, Intel Core I7 @3.40 GHZ and Windows 7 Professional. The android device that will be used for testing is a Sony Xperia XA Ultra: 3 GB RAM, 4X 2.0 GHZ 4X 1.2 GHZ ARM Cortex-A53, 8 cores and android 6.0.

4.1 Generic Offloading VS. Direct Offloading



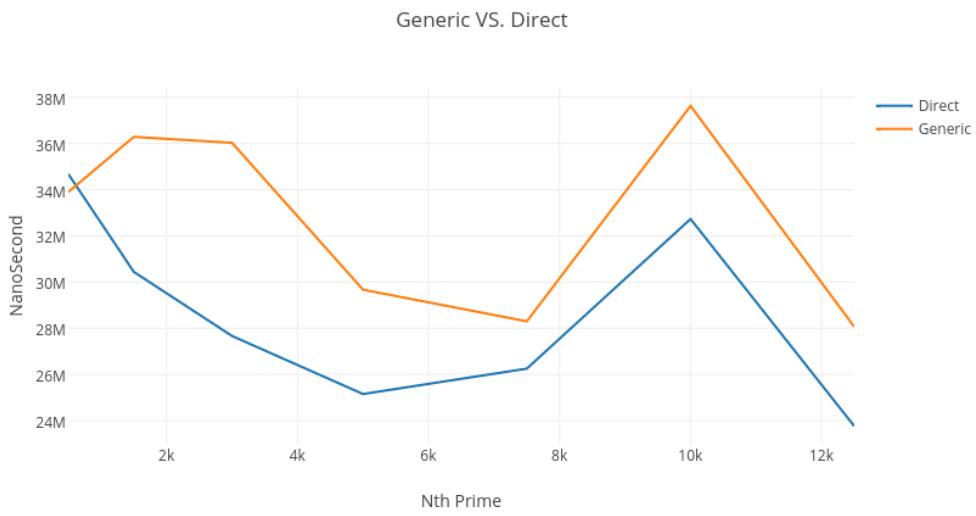


Fig. 2 Generic Offloading VS. Direct Offloading

In this section, Generic offloading runtime is compared to direct offloading. Figure 2 shows the runtime of the three Math algorithms using generic and direct offloading. One can see that the result of both methods is close and that sometimes the generic offloading is faster than the direct one. The differences in the runtime is very small since the runtime is measured in nanoseconds.

The main reason for this result is that there are only 3 generic offloaded methods that are on the server. This means that reflection will not affect the runtime massively since the class that contains the offloaded methods is relatively small. However, once there are more generic offloaded methods on the server the runtime of the generic model could rise significantly. This result presents an interesting idea of trying to always keep the classes that contain the offloaded methods small. However, that might lead to the model not being generic as it would require the model to know the name of the classes that the offloaded methods exist in. Reflection can still be used

to search for the specific classes but this will add more overhead runtime to the program to make it generic and it might be better to have one large class that contains all the offloaded methods.

4.2 Offloading VS. Non-Offloading (Local)

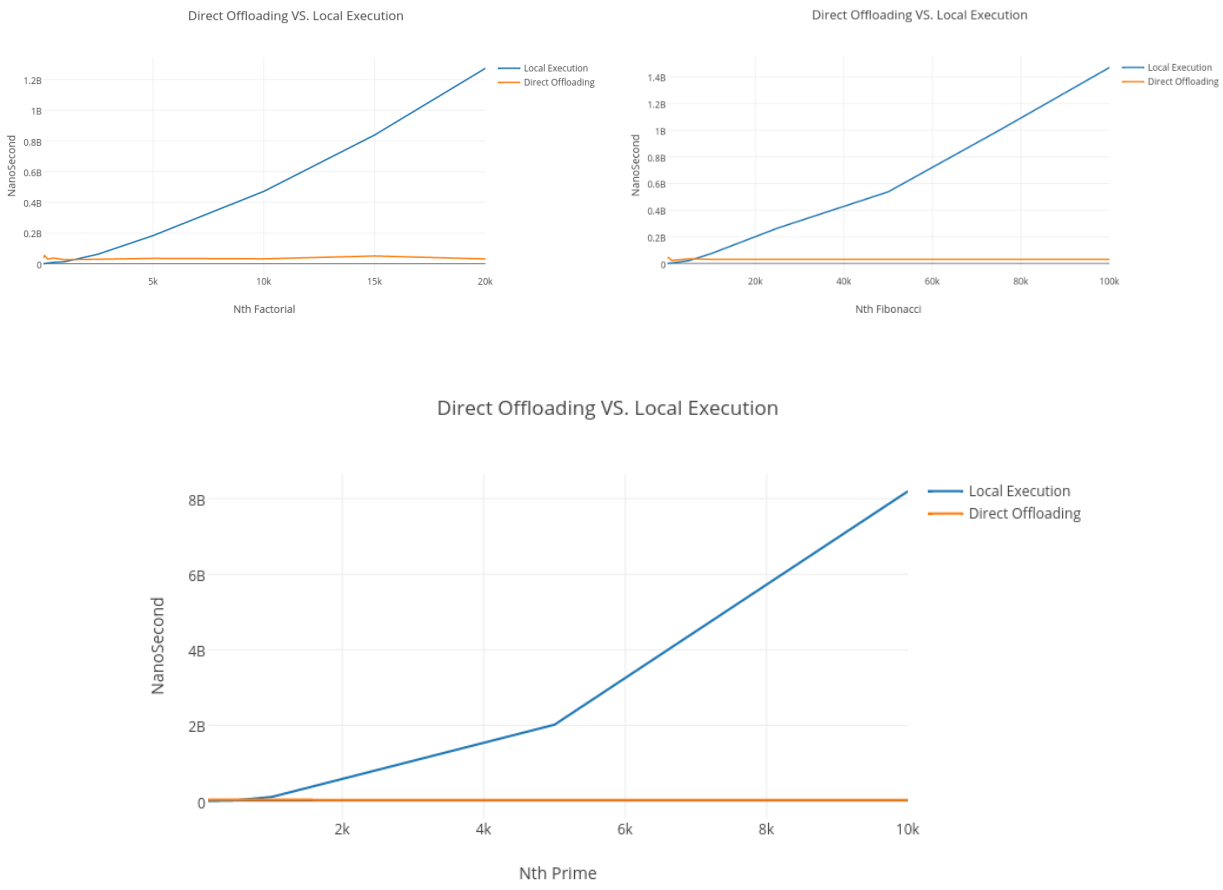


Fig. 3 Math Application, Direct Offloading VS. Local Execution

The major result from this paper is the offloading model (Direct or generic) VS. local execution on the smartphone. The results that is usually seen in offloading research papers is that the offloading model improves the performance, runtime and the battery of smartphones. Figure 3 is consistent with the results that are found in

other offloading models. The local execution on the android smartphone starts fast and it is much better compared to the offloading method, but once the input values get larger and larger, one can start seeing how offloading the code produces a speedup factor that seems limitless. One can observe that in figure 3 all three algorithms have the same pattern in their graph where the offloading starts off worse than the local execution but it then becomes much better in its runtime. This is due to the cost of transmitting the data to the server which is an overhead for small operations.

One can also observe that for each of the three algorithms, at a specific input the runtime on the smartphone starts increasing linearly for bigger inputs where the offloading runtime appears to be constant even with larger values. At certain inputs the application also crashes or freezes because the hardware for the smartphone is not capable of running the code anymore but by doing offloading one can exceed the hardware limitations and perform the tasks for those large inputs.

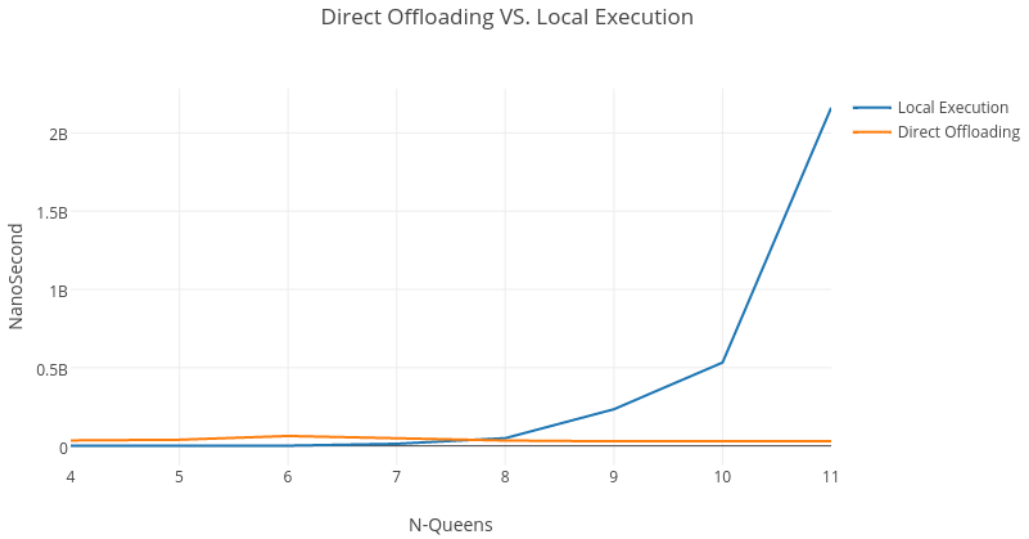


Fig. 4 N-Queens Puzzle, Direct Offloading VS. Local Execution

The N-Queens puzzle present an even more interesting result. Solutions for the puzzle start at n=4, but at the same time the smartphone was not able to execute the solution for more than n=11. This limits the graph above, but one can still see in figure 4 that at n=8 the local execution starts to increase. Unlike the Math application in figure 3 which grew linearly at some point, the N-Queens puzzle grows much faster after n=8, which is why it was only able to find the solution until n=11 and after that it would freeze. Again, one can notice the same observation for the offloading model that it appears like it stays constant compared to the runtime of the smartphone. This means that the offloading model for the N-Queens puzzle is very efficient since the smartphone can not handle small values like n=12.

5. CONCLUSION

Task offloading is a very promising topic with very interesting results that always seem to be good results but may be not good enough to work in every situation. Offloading is still in the beginning stages where it is not used that much outside of research topics but it can also help developers if they choose to use it. However, since there is no definite model for offloading, it makes it harder to use it and to make it as efficient as possible.

Generic offloading might be a way which allows developers to use the model without the need to build it themselves. MAUI presents an excellent generic model that does not require the developers to write a lot of extra code, but unfortunately MAUI focuses on energy consumption more than task offloading. The generic offloading presented in this paper has several requirements but it allows developers to offload code without building their own server or their own POST methods.

Direct offloading produced very good results in section 4.2. These results signal that offloading might not be too hard to accomplish with the new technologies. If more wireless networks become available publicly and if the mobile networks become faster, then offloading could be a major step towards bypassing the hardware limitations of android smartphones. The main purpose for the offloading model whether it is a generic one or not is to improve the performance of applications as well as allow the developers to create content that is not limited and that can match the level of content that is found on other powerful platforms like gaming consoles or PCs.

This paper showed that the offloading model produced positive results whenever the input value was big. This paper also presented a simple generic engine which developers can use to handle their offloaded methods without the need to write a POST method. For now, it seems like we need more hardware support to be able to make the hardware limitless.

5.1 Future Work

The generic offloading engine was planned differently at the beginning, so a lot of improvements can be made. The initial design for the generic model is that the developer can use it without the need to write or modify any code. This was not achieved and it might not be possible to achieve. Instead refining the current model to decrease its requirements as well as improving the reflection method in which the method is found and invoked is the next step in the model.

Determining the offloading at runtime is a popular method that several offloading models use to make their model more efficient. Adding such a feature to the direct and generic offloading models would be a big step towards making a model that developers can use. The offloading model in other papers calculates the execution time on the cloud/server and the transition cost to transfer the data to the server/cloud and then it compares these two costs to the cost of executing the method locally on the device, if the local execution is cheaper, then offloading the code is avoided. This is an improvement that would refine the offloading model that is currently in this project and make it more suitable to use.

Another improvement is to use nearby powerful machines to offload the code or even do the offloading themselves instead of sending the data over a mobile network that might be weak. If there are centralized powerful devices everywhere, then one can utilize the offloading model more by using those nearby machines to offload the code. These machines can also be connected to using a USB cable for example which would eliminate the need for a powerful network connection.

Finally, an OS offloading model could improve the efficiency and reduce the hardware limitations. An OS offloading model would monitor the user's usage and determine the offloading based on the user's behaviour instead of making calculations to decide whether to offload the code or not. This will allow for a faster offloading process as well as pre-offloading which means that based on the user's behaviour, the OS would offload the methods needed before the users use them.

References

[1] Gökçearslan, Şahin, Filiz Kuşkaya Mumcu, Tülin Haşlaman, and Yasemin Demiraslan Çevik. "Modelling Smartphone Addiction: The Role of Smartphone Usage, Self-regulation, General Self-efficacy and Cyberloafing in University Students." Computers in Human Behavior 63 (2016): 639-49. Web.

[2] Cuervo, Eduardo, Aruna Balasubramanian, Dae-Ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. "Maui." Proceedings of the 8th international conference on Mobile systems, applications, and services - MobiSys '10 (2010)

[3] PANKAJ. "Java Reflection Example Tutorial." JournalDev. N.p., 09 Oct. 2016. Web. 21 Apr. 2017.

[4] "Trail: The Reflection API." Trail: The Reflection API (The Java™ Tutorials). N.p., n.d. Web. 21 Apr. 2017.

[5] Barrameda, Jose, and Nancy Samaan. "A Novel Statistical Cost Model and an Algorithm for Efficient Application Offloading to Clouds." IEEE Transactions on Cloud Computing PP.99 (2015)

[6] Kosta, Sokol, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. " ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading." INFOCOM, 2012 Proceedings IEEE (2012): 945-53.

[7] "Queens.java." Princeton University. The Trustees of Princeton University, n.d. Web. 21 Apr. 2017.