ROSAC: A Round-wise Fair Scheduling Approach for Mobile Clouds Based on Task Asymptotic Complexity

Shahid Noor and Ragib Hasan
{shaahid, ragib}@uab.edu
Department of Computer and Information Sciences, The University of Alabama at Birmingham, AL 35294

***Abstract*—Mobile cloud computing has become popular because of its flexible infrastructure and easily accessible storage, sensing, and computing services. However, scheduling and allocating resources for the tasks provided by the cloud user are major challenges in the mobile cloud. The existing scheduling algorithms in the mobile cloud primarily focus on dividing and distributing a single task among multiple devices participating in the mobile cloud. However, when the load on the mobile cloud increases, multiple requests from different users can come at a single time point. The existing scheduling algorithms for distributed systems cannot be implemented in the mobile cloud, because the available resources differ extensively in the mobile cloud depending upon the time, geographic location, incentive policy, etc. While some cloud users' tasks are computed by their specified deadlines, the others suffer heavily. Therefore, we propose ROSAC, a fair scheduling approach based on the complexity of the cloud users' assigned task that minimizes the average delay between the task computation time and users' specified deadline. ROSAC works by round wise task division and distribution where the estimated time of all the subtasks in each round are the same. We design a simulated platform for a mobile cloud and compare the performance of ROSAC with some of the widely used scheduling approaches for distributed systems.**

***Keywords*- adhoc cloud, mobile cloud, cloud scheduling**

I. Introduction

Mobile Cloud computing provides users with a platform for performing resource intensive computations [1]. A primary concern for a mobile cloud user is finishing the user's outsourced tasks within the specified deadlines. While several scheduling approaches are available for efficiently provisioning a single task among the resources in the mobile cloud, none of them considers efficient scheduling of multiple tasks [2, 3]. With the growing popularity of the mobile cloud, a provider might receive multiple tasks at a single time point. The availability of diverse types of resources along with miscellaneous tasks in the mobile cloud often delivers unsatisfactory performance when the provider follows the traditional scheduling strategies, such as First-in-first-out (FIFO), Shortest-job-first (SJF), Round Robin or Priority Queue [4]. The resources in the mobile cloud are highly unpredictable since users frequently join, move, or leave a place. While some tasks always finish earlier than expected, some others always suffer to finish on time because of the allocation of an insufficient amount of resources. Therefore, it is necessary to provide an efficient algorithm for task scheduling in the mobile cloud so that all the tasks receive a fair amount of resources and also suffer uniformly during the limited availability of mobile cloud resources.

The first simplistic version of task scheduling used in Hadoop [5] was FIFO that allocated resource based on their arrival times. The biggest problem in FIFO is when a longer job comes first that eventually delays the processing time of other tasks in the queue. Hadoop MapReduce infrastructure introduced speculative task scheduling [6] where a simple heuristic function is used that compares the task progression with the average task progression and re-executes the tasks that have less progression than the average progression. However, in the heterogeneous mobile system, some devices have relatively lower computing power than some others. Therefore, their relative performance for computing a task is unsatisfactory compared to the other high-performance devices in the mobile cloud [7]. For improving the response time, Longest Approximate Time to End (LATE) was proposed, that considers progression rate instead of simple progression score [8]. Some researchers proposed reactive scheduling approaches where some tasks are migrated on demand for efficiently utilizing the resources and reduced the overall job completion time [9]. However, task division and distribution in a mobile device during the runtime is very expensive [10].

There are several challenges associated with scheduling tasks in the mobile cloud; first, selecting features for determining the order of the tasks in the queue is very challenging. Second, finding the free mobile devices appropriate for any task is non-trivial. Third, determining the appropriate amount of resources to be allocated for a task is hard. Finally, dividing a task into subtasks and distributing them efficiently among the allocated resource that would provide maximum throughput is tough.

In this research work, we propose ROSAC, a **RO**und-Wise pro-active **S**cheduling approach for mobile cloud based on the **A**symptotic **C**omplexity of the received task. ROSAC maximizes the resource utilization and reduces the average completion time all the accepted tasks by dividing and distributing the tasks fairly based on their complexity and the computing power of the mobile devices in the cloud. Initially, the mobile cloud retrieves the asymptotic complexity C_i of all the received task T_i, uses that information with its pre-computed task profile, and determines a $T_i' \subset T_i$ where the estimated runtime of all the T_i''s are the same. In each round, only T_i''s are considered for computation. All the T_i's are divided further into subtasks and distributed among the devices in their allocated resource pool based on the computing power of those devices. Once all the tasks in a round are completed, the mobile cloud repeats the same procedure for the remaining portion of all the unfinished tasks T_i.

Contributions: The contributions of this paper are as follows:

1) ROSAC is the first scheduling approach for mobile cloud that considers multiple tasks.
2) We provide a novel asymptotic complexity based task division scheme for efficiently dividing tasks.

3) We propose three different strategies for estimating the task computation time and allocate resources fairly for all the received task based on that estimation.

The rest of the paper is organized as follows: We discuss some of the related works in section II. We provide the overview of ROSAC in section III. The detailed operational model and the complexity are discussed in section IV and V respectively. We present our experimental results in section VI. Finally, we discuss the findings and conclude in section VII.

II. Related Work

Researchers have used several parameters for efficiently performing scheduling operations in the traditional cloud. Kristensen et al. proposed Scavenger, which distributes a task to a free surrogate that shows minimum computation and communication estimation time [11]. For common language, Shaw proposed timing schemes for measuring the maximum and minimum execution times [12]. However, their approach is based on global low-level attributes, which is difficult to integrate. Puschner et al. proposed WCET algorithms for integer linear programming (ILP) [13]. One major problem in their proposed approaches is that, a program needs to be converted into ILP to determine its WCET. Gustafsson presented a WCET estimation algorithm for object-oriented programming languages [14]. There are several data flow analysis (DFA) based approaches that estimate the running time based on the number of virtual method calls [15, 16]. Most of the DFA based methods considers relatively larger scopes during information computation. However, all the above work might fail to perform satisfactorily in the mobile cloud because of highly unpredictable nature of the mobile computing devices.

In the traditional cloud system, online task migration is a very popular approach for enhancing the efficiency while accessing the content or computing any task [17]. Similarly, in grid computing, it is noted that task migration can reduce the task computation time significantly because it enhances the overall resource utilization [18]. However, in the mobile cloud, on-demand migration becomes very expensive as it increases the cost associated with some additional computation and data transfer. For distributed systems, Rasley et al. proposed an efficient queue management technique considering several parameters, such as queue size, queue reorder, starvation freedom, and task placement to queues for enhancing efficiency during task computation [19]. The above approach can not be implemented in the mobile cloud because it is hard to ensure some dedicated amount of resources in the mobile cloud. In mobile cloud, Shah et al. proposed a method considering the history of a task and user mobility patterns, and dependency types within that task during task scheduling among mobile nodes to reduce the communication cost [2]. Li et al. designed several online and batch scheduling heuristics based on various user and system-centric parameters, such as makespan, waiting time, slow down, and resource utilization [3]. However, both of these approaches work efficiently only for a single task.

III. Overview of ROSAC

The high level structure of ROSAC is depicted in figure 1. Client initially contacts the cloud service provider for its task. Cloud Central Unit (CRU) is the central part of the cloud

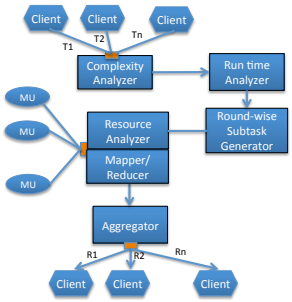


Fig. 1: **ROSAC Overview**

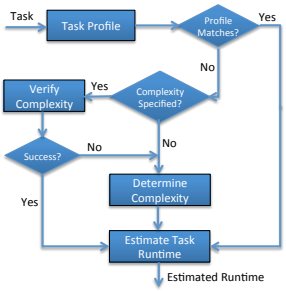


Fig. 2: **Estimating runtime based on task complexity**

system that receives tasks from clients. The complexity analyzer inside the CRU analyzes the task and determines its asymptotic complexity. The runtime analyzer inside CRU collects the complexity value, retrieves the profile of the task stored in its database, and determines a runtime estimation curve that species the estimated task computation time for various sizes of input. The round-wise subtask generator collects the estimated runtime information for all the tasks, compares those estimated run time to identify their relative complexity and based on that determines a subtask from the original task where all the selected subtasks have same relative complexity. The subtasks are packaged and sent to the resource analyzer for allocating resources to them. Resource analyzer collects the available resource information, assigns a resource pool for each subtask, and determines how the tasks should be divided further and distributed among the mobile unit (MU) inside its allocated resource pool. The mapper distributes each of those divided subtasks to its corresponding allocated MU. The reducer collects the result from all the MU and sends the reduced result to the aggregator. Once all the tasks in the first round are computed, the resource analyzer receives the second round tasks from the round-wise subtask generator and the process continues until a task is finished completely. The aggregator reduces the result obtained for each subtask of a task in its corresponding round and delivers the final result after reducing all the portion of the task.

Initially, when the task is received, CRU verifies whether it has a previously created profile for that task in its database or not. The profile of each task is composed of the task and its computation time for different sizes of input. When a task is computed in the cloud, CRU creates a profile for that task based on its completion time on a device. From the created profile, CRU estimates the task completion time for the given task. However, if no profile is found for the task then CRU retrieves the user specified complexity value from the task field. The complexity value is sent to the Complexity Verification Engine (CVE) and the output of the CVE specifies the correctness of the user specified complexity. In case the CVE finds incongruity in user specified complexity or fails to verify the complexity, CRU sends the source code of the task to the Complexity Detection Engine (CDE) that output the possible complexity value. Once the complexity is determined either by CVE, these complexity value is used for estimating the possible task computation time.

A. Round-Wise Task Division Algorithm

Initially, CRU determines the least complicated tasks (T_{min}) from the estimated runtime of each of those tasks. Next, CRU determines the relative complexity of each task, T by dividing

Algorithm 1 Round-Wise Task Division Algorithm

```
1: procedure TASKDIVISION(listOfTask< Complexity, InputSize >, listOfResource)
2:     var round=1
3:     while listOfTask!=null do
4:         var minComplexTask=getMinTask(listofTasks)
5:         var baseCompletionTime=minComplexTask.getCompletionTime()
6:         for curTask in listOfTask do
7:             var subtaskSize=baseCompletionTIme/
               curTask.getCompletionTime()*curTask.getRemainingSize()
8:             taskListByRound[round].add(task.getSubtask(subtaskSize))
9:             curTask.updateSize(subtaskSize)
10:            if curTask.getRemainingSize()=null then
11:                listOfTask.remove(curTask)
12:            end if
13:        end for
14:    end while
15: end procedure
```

its estimated completion time to T_{min}. If the length of T is l, then CRU determines a subtask T' of T where the size of T' is $C * l$. In another word, the estimated runtime of T' and T_{min} is the equal. Finally, T' is scheduled to its allocated resources and the size of T is updated once T' is computed. T is deleted from the task queue if its size becomes 0.

B. Round wise Resource Allocation

In round wise resource allocation process, the subtasks obtained in each round are divided and distributed further among the mobile devices based on their computing power so that all the divided portions of the subtask take approximately same time to finish in their respective devices. However, finding a optimal portion of the tasks for each node individually so that each node will take approximately same time is nontrivial as it is dependent upon the total number of mobile devices, the computing power in each devices, complexity of the task, and the size of the task. Our round-wise resource allocation approach is depicted in algorithm 2.

Algorithm 2 Reource Allocation Algorithm

```
1: procedure ALLOCATERESOURCE(task,listOfAssignedDevices)
2:     var slowestDevice=findSlowestDevice(listOfAssignedDevices,
3:                         task.getSize())
4:     var worstTime=slowestDevice.getExecutionTime(task.getSize())
5:     slowestDevice.setTask(task.getSize)
6:     DIVIDETASK(slowestDevice,listOfAssignedDevices,worstTime)
7: end procedure
8: function DIVIDETASK(task,listOfAssignedDevices,selectedDeviceList,
   referenceDevice)
9:     if selectedDeviceList.size=listOfAssignedDevices.size then
10:        return
11:    end if
12:    var referenceTime=referenceDevice.getExecutionTime(task.getSize())
13:    var subtaskSize=referenceDevice.getSize(referenceTime/2)
14:    referenceDevice.setTask(subtaskSize)
15:    var selectedDeviceList.add(referenceDevice)
16:    var remainingSize=task.getSize()-subtaskSize
17:    var minDiff=MAX_DOUBLE
18:    for curDevice in listOfAssignedDevices do
19:        if !selectedDeviceList.contains(curDevice) then
20:            var executionTime=
                curDevices.getExecutionTime(remainingSize)
21:            if executionTime-referenceTime/2< minDiff then
22:                minDiff=executionTime−referenceTime/2
23:                selectedDevice=curDevice
24:            end if
25:        end if
26:    end for
27:    selectdDevice.setTask(remainingSize)
28:    selectedDeviceList.add(selectedDevice)
29:    DIVIDETASK(referenceDevice,selectedDeviceList,referenceTime/2)
30:    DIVIDETASK(selectedDevice,selectedDeviceList,referenceTime/2)
31: end function
```

Our algorithm works in binary resource selection process. Initially, from the devices' profile lists, we choose the device that performs the worst for computing the given task. We consider that device as our reference device and the corresponding task computation time as our reference time. Next, we set up a target of reducing the worst time to half of the reference time by selecting another device from the device list by calling divideTask procedure (algorithm **??**). We analyze the reference device's profile and determine the size for which the reference device can compute the task approximately half of the reference time. We refer this subtask as reference task. Next, we observe each devices' profile and determine a device that can compute the remaining size of the task by approximately half of the reference time. We refer this device as selected device and the task as selected task. We set our reference time to the half of the current reference time and repeat the process for both the reference and selected task by calling divideTask procedure recursively until all the devices are taken.

IV. GENERATION OF THE RUNTIME CURVE

In this section, we will discuss all the three ways we will estimate the runtime in ROSAC.

A. Exact Profile Match

This approach can only be used when the CRU has some partial knowledge of the runtime information of any task, such as how long a device takes to run a given tasks for different sizes of input. Based on the different task sizes and their corresponding run time, CRU generates a polynomial function f following a curve fitting strategy where the degree of the polynomial corresponds to the asymptotic complexity of the task. If the asymptotic complexity of a task T is C, then the curve fitting algorithm generates the function f as,

$f(n, C) = a_C C + \sum_{l=0}^{p} a_l n^l$ where $l^p < C$

For example, if the asymptotic complexity of a task in O(nlgn). Then the generated function will be,

$f(n, nlgn) = a_{nlgn} + a_1 n + a_0$

The curve fitting algorithm determines the coefficient values from the task profile. On the other hand, if CRU has full knowledge for all the tasks, then it knows the task computation time for individual mobile devices inside the cloud system for some of the sample sizes of input. Since different mobile devices behave differently based on their configuration, distributing tasks based on the function generated output of the partial knowledge might give us a very different overall task computation time than our actual estimation. According to the full knowledge, CRU derives a device specific function for a task T with complexity C as,

$f(n, C, D) = a_{CD} C + \sum_{l=0}^{p} a_{lD} n^l$ where $l^p < C$

B. Demo Run

In this approach, we determine the relative complexity of all the received task by performing a demo run on a small portion of all the accepted tasks. Just like the exact profile match process, we represent the task computation time as a polynomial function f where the degree of the polynomial is the task's asymptotic complexity, i.e..

$f(n, C) = a_C C + \sum_{l=0}^{p} a_l n^l$ where $l^p < C$

However, since there is no pre-existed profile for the task unlike exact profile match, CRU determines the coefficient values from of the function by running a subset of the original task. Let us consider a sample task T of size S. Then we

retrieved $T_1 \subset T$ where size of T_1, S_1 is p% of S. CRU runs T_1 on its associated device and determines the task computation time. Next, CRU computes $T_2, T_3, .., T_p$ with size of $2 * S_1, 3 * S_1, ..k * S_1$ respectively, runs each of them and determines the task computation time. The value of p is dependent upon the degree of the equation. Suppose the complexity of a problem T of size n is represented as $C(T) = a_0 + a_1 * l$. Then our p value would be 2. We take two subtasks T_1 and T_2 of size S and 2*S from task T. Suppose the running time of T_1 and T_2 is t_1 and t_2 respectively. We get the following two equations,

$t_1 = a_0 + a_1 * S$ and $t_2 = a_0 + a_1 * 2S$

From the above two equations we get,

$a_1 = (t_2 - t_1)/S$ and $a_0 = t_1 - a_1 S$

We can know the values of t_2 and t_1 from the demo run. Once we get the values of a_0 and a_1, we can estimate the time t for running the complete task T. We can also get the coefficient for a task that follows a higher order equation, such as order of (nlgn), n^2. For example, if the degree of a complexity function of a task is n lg n then we get the following equation of time:

$t = a_0 + a_1 * n + a_2 * n * (lgn) =\approx a_1 * n + a_2 * n * (lgn)$

We can derive the coefficient values by running the problem with input size S and 2S as follows:

$t_1 = a_1 * n + a_2 * n * (lgS)$ and $t_2 = a_1 * n + a_2 * n * (lg2S)$

From the above two equations we get,

$a_2 = (t_2 - 2 * t_1)/(2 * S)$ and $a_1 = (t_1 - a_2 * S * (lgS))/S$

However, if we also want to compute the lower order coefficient a_0 then we need to find three subtasks T_1, T_2, and T_3 and run those to find the time t_1, t_2, and t_3, and then from there we can determine the coefficient values a_0, a_1, and a_2.

C. Similar Profile Match

According to this approach, CRU might not have any profile of the cloud user's provided task. However, it has the profile of a task that works approximately similar way. For example, suppose a cloud user wants to run a merge sort algorithm in cloud and CRU does not have any profile for the merge sort. However, CRU previously ran quick sort that works by divide and conquered process just like the merge sort algorithm and has the average asymptotic complexity of O(n lg n). CRU can use the task profile information of quick sort to predict the task computation time of merge sort. The accuracy of the computation of the user provided task is dependent on the task difference δ between the provided task and the task CRU picks from its database for profile matching. We can represent the difference between the task estimation following an exact profile match and similar profile match as follows:

$\delta = |(\sum_{i=1}^{S_{Tg}} n_i\gamma_i - \sum_{i=1}^{S_{Ts}} n_i\gamma_i)| * t_{Ts}$

Here, S_{Tg} and S_{Ts} are the total number of different statements in the given task Tg and similar task Ts respectively, n_i is the number of times the statement i is executed, γ_i is the normalized complexity of the statement i, and t_{Ts} is the estimated running time of Ts.

V. Complexity Analysis

Most of the time, the prediction of exact run time in seconds using static analysis method [20] provides incorrect results as different hardware behave differently under different circumstances [21]. Therefore, some researchers proposed time estimation based on total executed instructions. However, those approaches also fail because they overlook the time for accessing the cache or interruptions. Additionally, the parameters used in those methods need to be reconfigured when running to another machine because of the machine instructions and compiler optimization strategies use on that machine. Therefore, instead of delivering a scheduling algorithm based on the estimation of the task computation time, we focus on estimating the complexity analysis of a software or program based on the program flow diagram.

A. Graph Based Complexity Detection Engine

For determining the complexity, we represent a program as a complexity graph C-graph similar to a timing graph T [22] where the edges represent statement sequences of source code. The edges are weighted by the complexity associated with the execution of the statements corresponds to those edges. C-graph is represented as $G_c = (V, E)$ where V denotes the set of vertices and E denotes the set of edges where each $e_i \in E$ is represented by an order paired (v_i, v_j), such that $v_i, v_j \in V$. We assume that for every program there exists a specific starting and ending point i.e. there exist two vertices v_s and v_e that has no incoming and outgoing edge respectively. In addition, we assume that for each edge $e_i \in E$, there exists at least one path from v_s to v_e that contains e_i. Furthermore, the edges representing the complexity is always positive. An execution path in G is formed by an edge sequence from the starting to the ending vertex. We assume that the number of edges for each such execution path is bounded. The relative complexity of all the statements is pre-computed following our Single Statement Complexity algorithm. We determine an edge value from our pre-computed complexity value for the statement associated with that edge. We sum up all the edges values for getting the complexity of executing the specific execution path. Thus if $P = \lambda_1, \lambda_2, ...\lambda_p$ is an execution path, then the complexity of P, $\omega(P) = \sum_{i=1}^{p} \omega(\lambda_i)$.

During determining the path complexity, the edges can be taken in any order. Some of the edges might occur multiple times. Suppose the number of times an edge λ_k is specified by the function f_k and $P' = \lambda_1, \lambda_2, ...\lambda'_p$ is the set of all unique edges in P. Then using the following equation we get the path complexity, $\omega(P) = \sum_{i=1}^{p'} f_k \omega(\lambda_i)$

Since the total number of distinct execution path is countable, the average case complexity can be determined once we get the complexity for all the execution paths after dividing the total complexity value by the total number of possible paths.

B. Single Statement Complexity

Finding the complexity of individual statements in a programming language are difficult as there can be numerous statements each with different amount of complexity. For example, in Java there are 204 bytecode instructions. We label the complexity of each statement a value ranging from 0 to 1 where 0 and 1 indicate the least and most complicated statements respectively. A hash table is used for mapping a statement to its complexity value. The complexity value for each statement is precomputed and updated using algorithm 3.

We use an universal computing device (UCD) for computing the execution time of an instruction. When a statement inside a program is incurred, the corresponding complexity is retrieved from the hash table. If a new statement is arrived whose

Algorithm 3 Complexity Determining Algorithm

```
1: procedure FINDCOMPLEXITY(statement,maxTimeToExecute)
2:     var timeToExecute=getStatementExecutionTime(statement)
3:     if timeToExecute¡=maxTimeToExecute then
4:         complexityOfStatement=timeToExecute/maxTimeToExecute
5:         listOfComplexity.put(statement,complexityOfStatement)
6:     else
7:         complexityOfStatement=1
8:         for curStatement in listOfStatements do
9:             var curComplexity=listOfStatements.get(curStatement)
10:            var curComplexityUpdated=curComplexity*
                   timeToExecute/maxTimeToExecute
   listOfComplexity.put(statement,curComplexityUpdated)
11:        end for
12:        maxTimeToExecute=timeToExecute
13:    end if
14: end procedure
```

complexity value has not been computed yet, the corresponding statement is executed on UCD. If the execution time of that statement is lower than the maximum execution time, then the time value is the complexity of that statement. Otherwise, the complexity of the current statement is 1 and the complexity of all the other statements of the hash table is updated according to the new maximum time and complexity value.

VI. Simulation Model

A. Simulation Setup

We used Cloudsim for simulating the mobile cloud, which is a java based simulator for cloud computing [23]. We assumed that we can have five different types of devices Google Nexus 4,5, and 7 and Samsung Galaxy s4,s5. We created task profile for each device for 5 different tasks; word count, merge sort, inverted index, adjacency list, and sequence counter. We used the Java built-in PolynomialCurveFitter method where the higher order of the polynomial is the degree of the tasks' asymptotic complexity. We represented a cloudlet in CloudSim considering the task type and input size.

We considered a standard MacBook Computer as CRU and assumed that it communicates with the mobile nodes using the standard LTE system [24]. We designed the LTE using NS-3 simulator where we used the LTE's standard packet size of 1460 bytes. Each of the three sectors of an Enb nodes in LTE has 3.3 Gbit/s data transfer rate and has a coverage area of 60 meters [25]. The maximum transmission unit (MTU) is set to 1500 bytes, and the propagation delay is set to 10 ms whereas the packet interval delay is set to 6 ms.

For simplicity, we assumed that we know the asymptotic complexity of all the 5 problems. We represented the word count, inverted index, and sequence counter problem regarding the file size. For example, the complexity of a linear problem word count and sequence counter of file size 1KB is O(1024) and a linear problem inverted index with an input of 10 files of size 1 KB each is defined as O(10*1024). On the other hand, the input size of the merge sort and adjacency lists were represented in terms of the total number of integers in the input file. If n integer numbers are stored in the file then the asymptotic complexity of merge sort is O(n lg n). The complexity of an adjacency list program is linearly proportional to the total number of vertices and edges, i.e. O(n) when we represent the graph in a file as ordered pair p q of vertices.

For creating a curve for estimated time using Demo Run, we ran 5%, 10% of the sample problem word count, inverted index, adjacency list, and sequence count problems (since they are linear) and 5%, 10%, and 15% for the problem sorting (since its complexity is n lg n) just once. For example, if cloud receives 2 word count problem of size 1MB and 2MB respectively, we run the first word count problem with input size 50 KB and 100 KB and created an estimated run time profile based on that. For creating a curve following similar profile matches, we ran a known linear problem, Min-Max, where we would like to find the minimum and maximum values from a set of integer values. Our Min-Max problem consisted of 10 statements. We assumed that for any linear problem with size n, if s is the number of source code statements and for the same size Min-Max takes t ms time to finish, then our give problem will take t*s/10 ms. Similarly, for merge sort problem, we considered quick sort as our sample similar match problem. On the other hand, the exact profile matching algorithm process, we generated the time estimation curve after running each problem in our 5 different types of mobile devices several times with the various sizes of input. In Random strategy, we picked any unfinished task randomly until executed it completely and repeated the procedure until the task queue became empty. In Shortest Job strategy, we assumed that we know the completion time of a job in a standard device and sorted the task in ascending order based on their completion time. We executed the tasks in the sorted queue sequentially until the queue became empty.

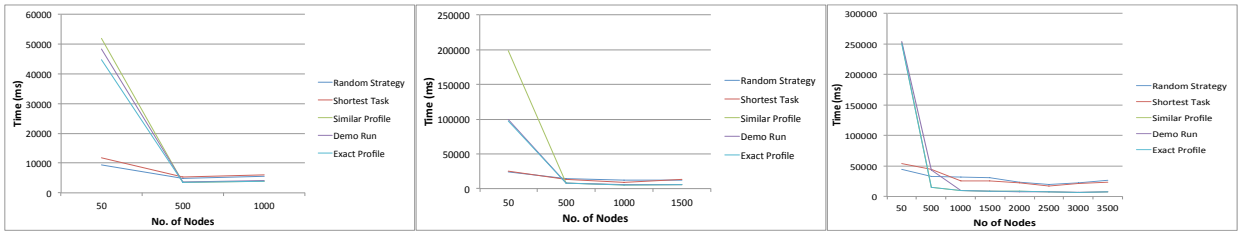
B. Experimental Results

In first experiment, we assumed that the input size of all the received task does not vary a lot. We considered that the longest task in each category is only 10 times larger than the smallest task for each type of tasks. Initially, we ran our simulation 10 times for 10 tasks and 50 nodes, and measure the average task completion time for all the five strategies. Next, we gradually increased the total number of mobile nodes until the task computation time reached the optimal point and measured the average task completion time. We repeated the whole procedure for the total tasks of 25 and 50. The corresponding figures are shown in figure 3a, 3b, and 3c respectively.

From figure 3a, 3b, and 3c we see that for a large number of mobile nodes all our three proposed strategies perform better than the random and shortest task strategies. However, the improvement is not significantly large when the task sizes are almost equal. Moreover, we can see that there is an optimal value for the total node and after that, the average completion time increases with the increasing value of the total nodes.

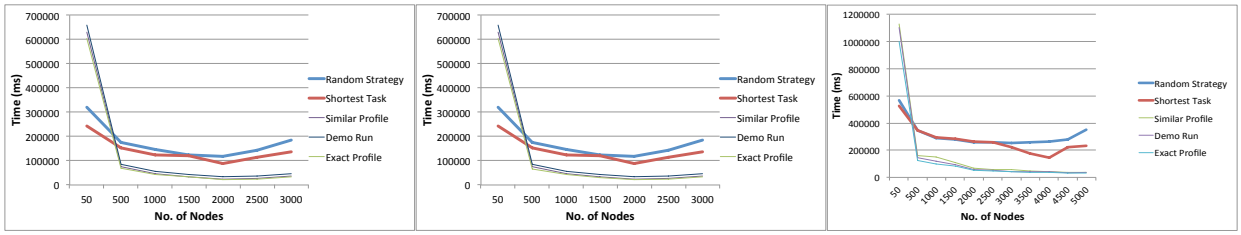
In our second experiment, we assumed that the input size of all the received tasks differs a lot in the size. We considered that the smallest task is 1000 times smaller than the largest one. We repeated the whole procedures of our first experiment and measured the task computation time for the different configuration of the cloud system. The corresponding figure is shown in figure 4a, 4b, and 4c respectively.

From figure 4a, 4b, and 4c, we see that the for small number of nodes, all our three proposed strategies' performances are not good and require more number of mobile nodes to finish the task before the time taken by the random or shortest task strategies. The reason is, each of our proposed strategies requires to run some decision algorithms before scheduling the actual task. This pre-computation is time-consuming considering the actual run time of a small number



(a) Node Size=10 (b) Node Size=25 (c) Node Size=50

Fig. 3: **Variation of average completion time with Node Size for Similar length Tasks**



(a) Node Size=10 (b) Node Size=25 (c) Node Size=50

Fig. 4: **Variation of average completion time with Node Size for Variable length Tasks**

of tasks. However, for the larger number of tasks, all our proposed strategies outperforms than the random and shortest task strategies in terms of the average completion time. The reason is that, for a large number of total tasks, random order strategy suffers because the larger tasks occupy resources for longer times and shortest task strategy suffers because the time associated with dividing a shorter task unnecessarily into more number of parts and the reduction of each of those parts' computed result is very large. From our experimental result, we also notice that scheduling based on exact profile match works best under every circumstance than the other two proposed strategies. On the other hand, between the other two proposed strategies, similar profile matches provides better performance for smaller number of tasks. However, for larger number of tasks, Demo run performs better than similar profile. The reason is that, in Demo Run, we ran certain percentages of every different types of tasks at least once for run time estimation curve generation. Therefore, the Demo run strategy is time consuming for the small number of tasks.

VII. Conclusion

In this paper, we proposed a novel scheduling approach for mobile clouds based on the asymptotic complexity of tasks. We proposed three different strategies for generating a curve for estimating the task completion time as part of the scheduling process in ROSAC. Moreover, we presented two approaches for determining the asymptotic complexity of tasks. From our experimental results, we showed that ROSAC based on any of our three proposed strategies outperforms the traditional random or shortest job strategies in terms of the average task completion time. ROSAC performs better when there is a significant difference in sizes among the tasks received by the mobile cloud. From the experimental results, we can also see that among the three strategies used in ROSAC, exact profile match is the best. In our simulation setup, we did not use any of our two proposed complexity determination approaches. Complexity determination might add some extra time in the overall task completion time. However, the task completion time for a large number of tasks would be still much lower than the traditional approaches especially for diverse task sizes. As a future work, we would like to deploy ROSAC in a real-world mobile cloud system and evaluate its performance.

VIII. Acknowledgements

This research was supported by the National Science Foundation CAREER Award CNS-1351038.

References

[1] S. A. Noor, R. Hasan, and M. M. Haque, "Cellcloud: A novel cost effective formation of mobile cloud based on bidding incentives," in *IEEE Cloud*, 2014.

[2] S. C. Shah, Q.-U.-A. Nizamani, S. H. Chauhdary, and M.-S. Park, "An effective and robust two-phase resource allocation scheme for interdependent tasks in mobile ad hoc computational grids," *Journal of Parallel and Distributed Computing*, 2012.

[3] B. Li, Y. Pei, H. Wu, and B. Shen, "Heuristics to allocate high-performance cloudlets for computation offloading in mobile ad hoc clouds," *The Journal of Supercomputing*, 2015.

[4] B. A. Shirazi, K. M. Kavi, and A. R. Hurson, Eds., *Scheduling and Load Balancing in Parallel and Distributed Systems*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1995.

[5] "Welcome to apache™ hadoop®!" http://hadoop.apache.org/.

[6] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *OSDI*, 2008.

[7] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel data processing with mapreduce: A survey," *SIGMOD*, vol. 40, no. 4, 2012.

[8] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *OSDI*, 2008.

[9] M. Hossain, S. A. Noor, D. Mukhopadhay, R. Hasan, and L. Li, "Cacros: A context-aware cloud content roaming service," in *SmartCloud*, 2016.

[10] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless Communications and Mobile Computing*, vol. 13, no. 18, 2013.

[11] M. D. Kristensen and N. O. Bouvin, "Scheduling and development support in the scavenger cyber foraging system," *Pervasive and Mobile Computing*, 2010.

[12] A. C. Shaw, "Reasoning about time in higher-level language software," *IEEE Transactions on Software Engineering*, vol. 15, no. 7, 1989.

[13] P. P. Puschner and A. V. Schedl, "Computing maximum task execution times — a graph-based approach," *Real-Time Systems*, vol. 13, no. 1, 1997.

[14] J. Gustafsson, "Analyzing execution-time of object-oriented programs using abstract interpretation," Ph.D. dissertation, May 2000. [Online]. Available: http://www.iss.mdh.se/index.php?choice=publications&id=0268

[15] O. Shivers, "The semantics of scheme control-flow analysis," *SIGPLAN*, vol. 26, no. 9, 1991.

[16] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for java," *SIGPLAN*, 2000.

[17] N. Tran, M. K. Aguilera, and M. Balakrishnan, "Online migration for geo-distributed storage systems," in *USENIXATC*, 2011.

[18] D. Choffnes, M. Astley, and M. J. Ward, "Migration policies for multi-core fair-share scheduling," *SIGOPS*, vol. 42, no. 1, 2008.

[19] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao, "Efficient queue management for cluster scheduling," in *EuroSys*, 2016. [Online]. Available: http://doi.acm.org/10.1145/2901318.2901354

[20] J. Engblom and B. Jonsson, *Processor Pipelines and Their Properties for Static WCET Analysis*, 2002.

[21] B. K. Huynh, L. Ju, and A. Roychoudhury, "Scope-aware data cache analysis for wcet estimation," in *RTAS*, 2011.

[22] P. P. Puschner and A. V. Schedl, "Computing maximum task execution times — a graph-based approach," *Real-Time Systems*, vol. 13, no. 1, 1997.

[23] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw. Pract. Exper.*, 2011.

[24] I. F. Akyildiz, D. M. Gutierrez-Estevez, and E. C. Reyes, "The evolution to 4g cellular systems: Lte-advanced," *Physical Communication*, vol. 3, no. 4, 2010.

[25] C. Dalela, "Article: Prediction methods for long term evolution (lte) advanced network at 2.4 ghz, 2.6 ghz and 3.5 ghz," *CTNGC*, 2012.