

MALMOS: Machine Learning-based Mobile Offloading Scheduler with Online Training

Heungsik Eom, Renato Figueiredo
*Advanced Computing and Information Systems Laboratory
 Electrical and Computer Engineering
 University of Florida, Gainesville, Florida, USA
 {hseom, renato}@acis.ufl.edu*

Huaqian Cai, Ying Zhang, Gang Huang
*Laboratory of High Confidence Software Technologies
 School of Electronics Engineering and Computer Science
 Peking University, Beijing, China
 {caihq12, zhangying06}@sei.pku.edu.cn, hg@pku.edu.cn*

Abstract—This paper proposes and evaluates MALMOS, a novel framework for mobile offloading scheduling based on online machine learning techniques. In contrast to previous works, which rely on application-dependent parameters or predefined static scheduling policies, MALMOS provides an online training mechanism for the machine learning-based runtime scheduler such that it supports a flexible policy that dynamically adapts scheduling decisions based on the observation of previous offloading decisions and their correctness. To demonstrate its practical applicability, we integrated MALMOS with an existing Java-based, offloading-capable code refactoring framework, DPartner. Using this integration, we performed quantitative experiments to evaluate the performance and cost for three machine learning algorithms: instance-based learning, perceptron, and naïve Bayes, with respect to classifier training time, classification time, and scheduling accuracy. Particularly, we examined the adaptability of MALMOS to various network conditions and computing capabilities of remote resources by comparing the scheduling accuracy with two static scheduling policies: threshold-based and linear equation-based scheduling policies. Our evaluation uses an Android-based prototype for experiments, and considers benchmarks with different computation/communication characteristics, and different computing capabilities of remote resources. The evaluation shows that MALMOS achieves 10.9%~40.5% higher scheduling accuracy than two static scheduling policies.

Keywords—Mobile platform, computation offloading, machine learning, runtime scheduler, online training

I. INTRODUCTION

Over the last decade, mobile offloading techniques have emerged as a means to overcome resource constraints of mobile platforms (e.g. smartphones, tablets). Offloading allows these devices to delegate computationally-intensive tasks to more powerful external resources, such as personal workstations or cloud servers, with intent to improve application performance, while conserving energy consumption of mobile devices. Initially, research interests on mobile offloading techniques have focused on core mechanisms that deal primarily with *what to offload* and *how to offload*. The research community has studied various approaches to implement mobile offloading frameworks which fall in the following categories: application partitioning [1]–[3], thread migration [4], [5], application migration [6], and distributed

offloading frameworks [7], [8].

One important fact is that benefits and drawbacks from offloading computation-intensive portions of mobile applications can be influenced by various internal and external factors, such as application requirements, network conditions, and computing capabilities of mobile and external devices. Accordingly, *whether to offload or execute locally* needs to be decided dynamically based on internal and external conditions, and monitoring the aforementioned dynamic features at runtime is required to make offloading decisions. Otherwise, incorrect offloading decisions may cause performance degradation and/or increase energy consumption. Related work has also considered dynamic scheduling for mobile offloading frameworks. For example, Kwon et al. [9] consider a simple threshold-based scheduler in which the framework decides to offload mobile computations only when the data transfer size is higher than a certain threshold. MAUI [1] utilizes a linear regression model using predefined features, such as workload, network and device characteristics, to make offloading decisions.

Even though these systems consider runtime schedulers for mobile offloading, which take dynamic features such as data transfer size or network conditions into account to make offloading decisions, it is impractical for these approaches to build a generic offloading decision policy embracing all possible cases in dynamic mobile environments. For instance, consider that a mobile user walks around a shopping complex, while receiving offloading-capable mobile services such as gaming or augmented reality. In this situation, network latency, bandwidth and availability can be changed from moment to moment. In addition, according to network conditions, the offloading service may trigger the service migration to other remote servers (which may have different computing power capabilities) to deliver better service experience to the mobile user. It is infeasible to define a static scheduling model to account for various network conditions and computing capabilities of remote offloading servers. Furthermore, the scheduling policy, which is fitted for a particular application, may not work correctly for other applications due to different application requirements and

characteristics. For example, gaming and image processing applications may have different computation complexity, even though they process a similar size of data.

In this paper, we aim to develop a novel framework for an adaptive runtime scheduler for mobile offloading by employing online machine learning techniques – MALMOS (machine learning-based mobile offloading scheduler). In this approach, a machine learning classifier makes decisions of whether mobile computations should be offloaded to external resources, or executed locally. To this end, we extend our previous work on offline machine learning-based runtime scheduler [10], and develop a novel online scheduling module in which any appropriate machine learning classifier can be utilized for the runtime offloading scheduler. Furthermore, MALMOS provides an online training mechanism for the machine learning-based runtime scheduler such that it supports a policy that dynamically adapts scheduling decisions at runtime based upon the observation of previous offloading decisions and their correctness.

To demonstrate its practical applicability, we integrated MALMOS with an existing Java-based, offloading-capable code refactoring framework, *DPartner* [11]. Originally, the offloading-capable mobile applications generated by *DPartner* depend on static (or user-provided) input to decide whether to execute offloadable computations (i.e. Java classes) locally or remotely. By combining this paper’s machine learning-based runtime scheduler with these applications, offloading decisions are dynamic and do not require any user input. We have implemented an online-scheduled *DPartner* prototype, and used it to perform quantitative experiments with Android mobile devices and applications to evaluate the performance and cost for three machine learning algorithms: instance-based learning, single layer perceptron, and naïve Bayes, with respect to classifier training time, classification time, and scheduling accuracy. Particularly, we examined the adaptability of MALMOS to various network configurations and computing capabilities of remote resources by comparing the scheduling accuracy with two static scheduling cases: threshold-based and linear equation-based scheduling policies. Even though there have been prior related studies which suggest utilizing machine learning techniques for mobile computing environments, to the best of our knowledge, our work is the first to consider an online training mechanism for the machine learning-based runtime scheduler, and to demonstrate the system performance and cost of various machine learning algorithms for the runtime scheduler of mobile offloading tasks through an end-to-end experiment.

The rest of the paper is organized as follows. In Section II, we overview prior research efforts on mobile offloading frameworks as well as the use of machine learning techniques for scheduling problems from various domains. Section III summarizes our previous work which proposed the machine learning based runtime scheduler for mobile

offloading framework, and motivates the concept of the online training mechanism for MALMOS. In Section IV, we describe the architecture and main modules of MALMOS and explain our implementation details. Then, Section V evaluates the prototype implementation of MALMOS with respect of the scheduling performance and cost. Finally, we conclude the paper in Section VI.

II. RELATED WORKS

A. Adaptive Mobile Offloading Frameworks

Many approaches have been proposed to address mobile offloading scheduling problems on dynamic environments. In [12], a prediction heuristic uses linear functions to estimate the local and remote processing time as well as the data transfer time. The remote server updates these linear functions using least-squares method and returns the updated prediction functions to the mobile client, where these functions are used to compare the performance between local processing and offloading. Gu et al. [13] try to relieve the memory restriction of the mobile device by adaptively making offloading decisions with help from a fuzzy control model, called Offloading Inference Engine (OLIE). OLIE monitors the available memory margin of the mobile device and network bandwidth, and uses them to derive offloading decision specifications defined by the application developer so that when present condition matches the specified rule, the mobile workloads are offloaded to a remote server. Kwon et al. [9] consider a simple rule-based scheduler in which the framework decides to offload the mobile computation only when the data transfer size is higher than a certain threshold.

Although the aforementioned studies take dynamic features from hardware or network level (i.e. available memory size, network bandwidth) into account, they still depend on predefined static decision rules or cost models while preventing the scheduler from adapting to dynamic conditions at runtime. In contrast, our approach does not rely on any predefined specifications or prior knowledge of the mobile application. Instead, we consider machine learning techniques for the adaptive mobile offloading scheduler in which the scheduler can be trained on and dynamically make offloading decisions at runtime.

B. Machine Learning Techniques for Runtime Scheduler

Areas such as heterogeneous computing, grid computing, and data center management have used machine learning techniques to address dynamic scheduling problems. In [14], machine learning techniques are used to provide a compiler-based, automatic and portable predictor for multi-core processors. In order to decide the optimal number of threads and scheduling policy, the framework used a feed-forward artificial neural network and a multi-classes support vector machine model, respectively. Berral et al. [15] propose an energy-aware data center through server

consolidation by turning off idle servers with assistance from machine learning based scheduling. The scheduler predicts the future performance of the jobs and power consumption in the resulting job allocation using linear regression algorithms. This framework uses artificial neural networks for the performance modeler (which predicts task computation and data communication costs) and the modeler is used with a directed acyclic graph to determine an appropriate schedule.

III. BACKGROUND AND CHALLENGES

In this section, we summarize our previous work on offline-trained machine learning-based runtime scheduler for mobile offloading framework. Then, we describe challenges on the online training mechanism for the machine learning-based runtime scheduler.

A. Offloading Performance

In our previous work [10], we studied a runtime scheduler for mobile offloading framework through detailed measurement experiments. With an OpenCL-based mobile offloading framework [16], we performed various experiments using four different OpenCL workload kernels used in a variety of areas such as image processing and simulations. Also, in order to observe the impact of different network conditions on the offloading performance, we configured different network conditions: local area network, campus network, and wide area network (i.e. Amazon EC2 cloud). In the evaluation, we verified that different network conditions result in significant differences in offloading performance. Also, each OpenCL workload kernel shows different offloading performance, even though they process similar size of data, because each kernel has different computational complexity. These results demonstrated offloading performance variation over different network conditions and OpenCL workload kernels.

Based on our observation, we applied machine learning techniques to runtime scheduling. We trained various machine learning classifiers with *Weka*. Then, we implemented a machine learning-based runtime scheduler by building the trained machine learning classifiers onto the OpenCL-based mobile offloading system. In our evaluation, we observed that most of the machine learning-based schedulers show scheduling accuracy higher than 80%.

B. Offline vs. Online Training

There exist two possible ways to train the machine learning classifier: *offline* and *online* training. In offline training, the machine learning classifier can be trained using a set of pre-collected static data. Once the machine learning classifier is trained in the initial training phase, the classifier does not change its prediction behavior. It is therefore difficult to reflect unseen situations or conditions which have not been trained in the initial training phase. For that reason,

offline machine learning should be trained through a large set of data which covers as many cases as possible in order to accomplish high prediction performance.

On the other hand, the online machine learning technique does not depend on any pre-trained classifiers to predict the future behaviors of the target system. Instead, the online machine learning techniques trains its classifier when a new data instance is available at runtime. More specifically, the classifier of the online machine learning technique is trained whenever the comparison result between the prediction value and actual behavior is available. Thus, the main challenge of online machine learning is to deliver the prediction correctness into the training process, so that it can train the classifier continuously based upon the prediction correctness feedback. However, it can be too expensive to determine the correct prediction, since the scheduler may have to attempt all of the possible decision cases to know which prediction is correct.

In this paper, we extend our previous work on the offline machine learning-based mobile offloading scheduler by considering an adaptive online training mechanism. In the case for the mobile offloading scheduler, even though there exist only two possible cases (offloading and local processing), the mobile platform still has to pay for both offloading and local processing. Therefore, it is important to minimize the training cost while guaranteeing reasonably acceptable scheduling performance. We address this challenge by proposing the adaptive online training mechanism in which the training phase can be dynamically stretched and shrunk according to the scheduling performance so that the mobile device does not have to pay the unnecessary training cost by retaining a static length of the training phase.

IV. ARCHITECTURE OF MALMOS

In this section, we describe the architecture and main modules of MALMOS. Then, we describe implementation details on MALMOS with the adaptive online training mechanism. Lastly, we explain how MALMOS can be integrated with mobile offloading frameworks, using DPartner as a concrete scenario. The overall architecture and main modules of MALMOS are illustrated in Figure 1.

A. Architecture and Modules of MALMOS

As depicted in Figure 1, the overall architecture consists of four main modules: computation dispatcher, runtime scheduler, machine learning classifier, and the trainer for the machine learning classifier. These four main modules interact with each other to schedule and execute offloadable tasks, and to train the machine learning classifier. At runtime, MALMOS operates alternatively in either of two different phases: *runtime scheduling phase* and *online training phase*. In the runtime scheduling phase, tasks are executed locally or remotely based on the decisions from the scheduler. First, the computation dispatcher receives an offloadable

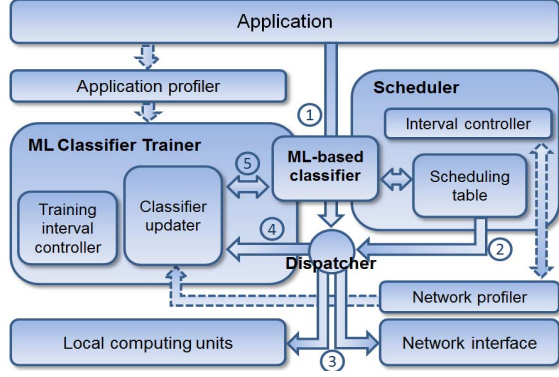


Figure 1. Overall architecture of MALMOS with online training. The dotted lines indicate the application and network features flow path, and the solid lines are the application computing tasks or scheduling-related commands flow path.

task (①). Then, according to the decision provided by the scheduler module (②), the dispatcher forwards this task to the appropriate execution unit (③): either a local computing unit, or a remote computing resource (through the network interface).

In contrast, in the online training phase, the machine learning classifier is trained in accordance with the scheduling performance (i.e. scheduling accuracy). In order to determine the scheduling performance, the offloadable computation task is forwarded to and executed in *both* execution units; the computation dispatcher records both execution times, and feeds back the performance comparison to the classifier trainer (④). Based on this feedback from the dispatcher, the classifier trainer updates the machine learning classifier (⑤). The implementation details for these modules are as follows:

Computation dispatcher. This module has the responsibility to dispatch and forward offloadable computation tasks to the appropriate computing unit: remote computing resource (for offloading) or local processing units (for local execution) according to the scheduling decision from the scheduler module stored in the scheduling table. In the online training phase, however, the computation dispatcher works differently by forwarding and executing computation tasks in both local processing units and remote computing resources. Also, this module provides feedback to the classifier update in the machine learning classifier trainer module, by comparing the performance between offloading and local processing and feeding back which execution leads to better performance.

Machine learning classifier. The machine learning classifier is in charge of making decisions on offloading or local processing for offloadable tasks. The main difference, which distinguishes MALMOS from the previous approaches on mobile offloading schedulers, is that it does not rely on any predefined scheduling policies. Instead, the machine

learning classifier hooks up to the classifier trainer, so the trainer updates the classifier at runtime whenever the feedback from the computation dispatcher is available, and the machine learning classifier adapts its decision behavior with changes of internal and external environments. When the scheduler module sends the request for offloading decisions with the attributes obtained by the network and application profiler, the machine learning classifier makes the decisions and stores them into the scheduling table in the scheduler module. Even though it is possible to adopt various attributes from internal (application) and external (network, remote resources) environments, for the current implementation, we employ two attributes: the size of data to be sent to the remote resource, and the network bandwidth.

Runtime scheduler. This module schedules the offloadable computation tasks by sending the decision requests with the capture of internal and external dynamic features. Then, it stores the decision results from the classifier to the scheduling table so that the computation dispatcher can forward each computation tasks to the appropriate computing unit through looking up the scheduling table. Scheduling the computation tasks requires to profile internal and external dynamic features, such as data size of inputs to the tasks and network conditions. We achieved this profiling by implementing application and network profilers. For the application profiler, we defined an application programming interface (API) to monitor the invocation of the offloadable computation tasks and capture the size of input arguments. Also, for the network profiler, we implemented network bandwidth measurement by simply dividing the size of a probing packet with the elapsed time to send the probing packet as described in [17].

However, profiling application and network conditions incurs additional cost in terms of runtime overhead and energy consumption, which means that too frequent profiling may lead to high overhead. For this reason, we consider two scheduling strategies. In the first strategy, each time offloadable tasks are invoked from the mobile application, the runtime scheduler requests to make offloading decisions while profiling application and network conditions. We refer to this strategy as *on-demand scheduling*. Another strategy is *periodic scheduling*, in which the scheduler makes the decision requests asynchronously with actual invocations of offloadable tasks, but periodically with a certain interval. Therefore, each profiler does not need to inspect application and network conditions for every invocation of the offloadable tasks. Also, in the periodic scheduling strategy, the scheduling interval can increase and decrease dynamically according to network behavior in order to obtain more up-to-date network conditions. If the variation of network bandwidth is less than a threshold, which can be thought as a steady-state network status, the scheduling interval becomes longer (e.g. twice as current interval). The assumption is that it is likely that former scheduling decisions can be still

```

1: // switch the training and runtime scheduling phase
2: while  $execution_{curr} < p_{curr}$  do
3:   if  $phase == SCHEDULE$  then
4:     do scheduling
5:   else
6:     if  $accu_{curr} > accu_{th}$  then
7:       switch to runtime scheduling phase
8:     else
9:       stay training phase
10:    end if
11:  end if
12: end while
13:
14: // calculate the length of the next period
15: if errors happened then
16:    $p_{next} \leftarrow MAX(p_{curr} - (p_{min} \times n_{err}), p_{min})$ 
17: else
18:    $p_{next} \leftarrow MIN(p_{curr} + p_{min}, p_{max})$ 
19: end if

```

Figure 2. Adaptive online training mechanism

effective for the near future. In contrast, if the variation of network bandwidth is greater than the threshold, the interval becomes half of the current interval, leading to more frequent scheduling.

Machine learning classifier trainer. With the feedback on the performance comparison between offloading and local processing from the computation dispatcher, the machine learning classifier trainer updates the machine learning classifier. In order to compare the performance between offloading and local processing, the trainer creates one separate thread for local processing, so that offloading and local processing can be executed concurrently. Then, the computation dispatcher measures the execution time for both offloading and local processing to compare the performance and provides the comparison result to the classifier updater. Finally, the classifier updater trains the machine learning classifier with the feedback from the computation dispatcher and the attributes from the profilers.

B. Adaptive Online Training Mechanism

One of the main contributions of this work is an adaptive online training mechanism which allows the training phase duration to stretch and shrink dynamically according to the scheduling accuracy. The pseudo-code of the adaptive online training mechanism is shown in Figure 2. First, lines 2–12 decide to switch the online training and runtime scheduling phase by calculating the current scheduling accuracy where $execution_{curr}$ is the current execution within one period and p_{curr} is the current length of one period. In the training phase, if the current accuracy ($accu_{curr}$) is higher than the accuracy threshold ($accu_{th}$), the online training mechanism switches to the runtime scheduling phase. Otherwise, it stays

in the training phase. Next, the length of the next period (p_{next}) is calculated according to the number of incorrect decisions within the training phase (lines 15–19).

Figure 3 illustrates an example of how the adaptive online training mechanism works. In this example, we set the accuracy threshold ($accu_{th}$) to 70% and the minimum length of one period (p_{min}) to 5 to illustrate a representative scenario. Also, we assume in this example that there is no incorrect decision or scheduling error in the first, second, and fourth period, but one error in the third period. In order to measure the current scheduling accuracy ($accu_{curr}$) of the classifier, the trainer observes whether the performance comparison matches with the offloading decision. For example, if the classifier makes a decision to offload, and offloading actually has better performance than local processing, we regard this case as a correct decision. Then, we calculate the current scheduling accuracy using Equation 1.

$$accu_{curr} = N_{correct} / (N_{total} + 1) \quad (1)$$

where $N_{correct}$ and N_{total} are the number of correct decisions and total decisions. We add one to the denominator in order to avoid 100% of scheduling accuracy at just one training (if the decision is correct in the first training turn, the scheduling accuracy becomes 100%).

Following the example, in Figure 3, the runtime scheduling phase begins after three training turns in the first and second period because the scheduling accuracy at third training turn is 75%, which is higher than $accu_{th}$ (70%). Furthermore, because there was no decision error in the first and second periods, the next period length increases by adding the minimum length of one period. Thus, the lengths of the second and third period become 10 and 15, respectively. In the third period, however, as one incorrect decision has occurred, the training phase holds for six executions when the scheduling accuracy becomes higher than 70%, and the length of the fourth period decreases by the minimum length of one period multiplied by the number of incorrect decisions (in this case, 1). This adaptive online training mechanism is inspired by the feedback control algorithm used for TCP congestion avoidance, *Additive Increase/Multiplicative Decrease*.

C. Integration with On-Demand Java-based Offloading Framework

To demonstrate the applicability of this approach, we integrated the modules of MALMOS into the DPartner mobile application framework, which is capable of automatically refactoring Java-based mobile applications to identify offloadable computation tasks. Given an Android mobile application, first of all, DPartner examines its bytecode to classify the Java classes into anchored and offloadable classes, so it can be guaranteed that the anchored classes are executed on the mobile device, for example to directly use a variety of the local sensors such as the GUI display, camera,

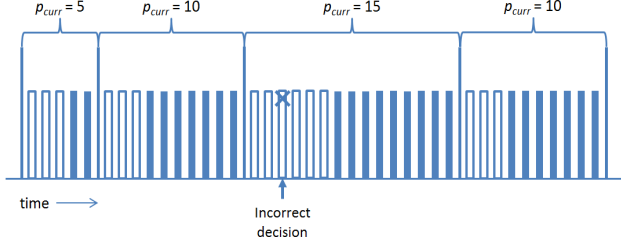


Figure 3. An example of the adaptive online training mechanism. Empty bars indicate the training phase and filled bars mean the scheduling phase. For this example, we set 70% of the scheduling accuracy threshold ($accu_{th}$) and 5 of the minimum length for one period (p_{min}). Also, p_{curr} is the current length of one period.

accelerator, and GPS. Also, DPartner rewrites the bytecode for the offloadable classes to implement a special type of program structure to support on-demand offloading. Then, DPartner packages files (such as images, xml files, external libraries, and rewritten bytecode) for the offloadable classes, and generates two separate objects which are deployed into the mobile device and the remote server, respectively.

With respect to scheduling, the offloading-capable Android applications generated by DPartner require either static or user-provided decisions. However, static scheduling becomes inaccurate if conditions change, and it is impractical that the mobile user monitors internal and external environments and schedules each offloadable mobile computations at runtime. We address this challenge by integrating MALMOS with the offloading-capable mobile applications generated by DPartner. We were able to achieve this integration without requiring deep modifications or code additions to DPartner. In fact, the only major code addition is to define new APIs to acquire the data size of offloadable computations and the network bandwidth. By combining MALMOS with these applications, the mobile user can be free of the burden of scheduling tasks.

V. EVALUATION

In this section, we evaluate the prototype implementation of MALMOS with respect to the scheduling performance and cost. First, we examine the training and classification time for different categories of machine learning algorithms: instance-based learning, single layer perceptron, and naïve Bayes. We examined the adaptability of MALMOS to various network conditions and computing capability of remote resources by comparing the scheduling accuracy with two static scheduling cases: threshold-based and linear equation-based scheduling policies, by running three synthetic applications (hidden Markov model, floating-point matrix multiplication, and sobelfilter) and two real applications (Linpack and Go game).

Table I
AVERAGE AND STANDARD DEVIATION OF NETWORK LATENCY AND BANDWIDTH FOR LOCAL AREA NETWORK, CAMPUS NETWORK AND WIDE AREA NETWORK.

	LAN		Campus network		WAN	
Latency (ms)	Avg.	Stddev.	Avg.	Stddev.	Avg.	Stddev.
	10.83	2.68	15.46	4.18	74.03	17.73
Bandwidth (Mbits/s)	Avg.	Stddev.	Avg.	Stddev.	Avg.	Stddev.
	52.18	1.41	19.69	1.90	1.42	0.18

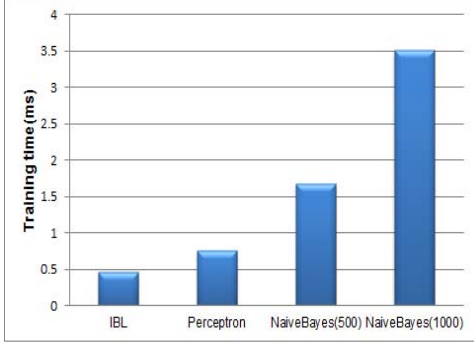
A. Experimental Setup

Even though high-dimensional machine learning algorithms (such as decision tree, multi-layer perceptron, or support vector machine) can achieve more accurate scheduling performance, they might be too expensive to be used for mobile platforms. Based on algorithm complexity considerations, we selected three machine learning algorithms: instance-based learning, single layer perceptron, and naïve Bayes for our prototype implementation.

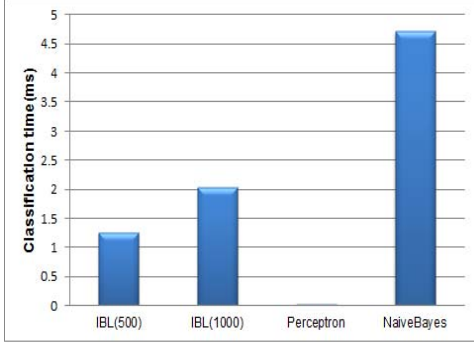
Also, in order to evaluate the adaptability of MALMOS to various network conditions and computing capabilities of remote computing resources, we setup experiments using a variety of hardware and network configurations. First of all, our hardware setup consists of a mobile client and three different remote servers. We have utilized an Android smartphone, Nexus 5 equipped with 2.3Ghz quad-core processor and 2GB RAM, and running Android KitKat as the mobile client. For the remote servers, we used two laptops and one desktop which have different levels of computing power capabilities. The first laptop has an Intel i5 Quad core 2.6Ghz processor and 8GB RAM, while the second laptop is equipped with an Intel Core2 Duo 2.0Ghz processor and 2GB of memory. In addition, we ran CPU-intensive workloads on the second laptop using *Stress*, a simple workload generator [18], to observe the effect of CPU load at the remote resource on the scheduling performance. The third remote resource is a workstation with an Intel Core2 Duo 3.0Ghz processor and 8GB RAM. All remote servers run Linux OS, Ubuntu 12.04.

Instead of installing the aforementioned three remote servers in different network environments, for the network configurations, we emulated three different network bandwidth characteristics: local area network, campus network, and wide area network by using the Traffic Control (TC) tool. The experimental setup using TC gives flexibility to vary network conditions between the mobile device and remote resources. Table 1 summarizes the average and standard deviation of network latency and bandwidth measurements that we setup for the experiments.

We experiment with three synthetic applications: hidden Markov model, floating-point matrix multiplication and sobelfilter, and two real applications available in Google Play: Linpack [19] and Go game [20], each having different application requirements and computation complexities. For



(a) Training time



(b) Classification time

Figure 4. Training and classification time comparison for three machine learning algorithms. The classification time for perceptron is 0.03ms.

example, sobelfilter is an image edge detection filter; it takes a relatively large size of input, while its computation complexity is low. Therefore, we classify sobelfilter as a communication-intensive application. In contrast, the core computation of Go game is a Monte Carlo algorithm with iterative random sampling tasks which have high computation complexity. Thus, Go game is a computation-intensive application. Otherwise, the computation complexity of other three applications depends on the size of input such as matrix size (for floating-point matrix multiplication and Linpack), and the number of states (for hidden Markov model).

B. Overhead of training and classification

Since our machine learning-based mobile offloading scheduler keeps training the classifier, which leads to runtime overhead, we measured the cost to train the machine learning classifier. As shown in Figure 4(a), instance-based learning has the shortest classifier training time among three machine learning algorithms. In fact, the basis for the classification of instance-based learning is the instance database, where previously experienced cases are stored. Instead of training the explicit classifier, therefore, instance-based learning adds a new instance to the database for the future classification, resulting in relatively small training overhead. On the other hand, naïve Bayes with 1000

instances of database has the largest training overhead. From the algorithm perspective, naïve Bayes measures the probability density for the given classes with the mean and standard deviation, and chooses the class with the highest probability. Thus, naïve Bayes calculates the mean and standard deviation throughout the previous instances for each class whenever a new instance is trained. Consequently, naïve Bayes with different number of instances incurs different training overhead, and in our experimental setup, naïve Bayes with 1000 instances, which requires relatively heavy calculations for the mean and standard deviation, has bigger training overhead than other machine learning algorithms.

Also, Figure 4(b) shows the classification time that the machine learning classifier consumes to make a decision. Similarly as the training time, naïve Bayes shows bigger overhead than other machine learning algorithms. We used the Gaussian distribution to calculate the probability density for each class with the mean and standard deviation value. Therefore, the classification for naïve Bayes entails complex arithmetic operations, such as the exponential function and the square root, which take longer than addition, multiplication, or comparison. For that reason, naïve Bayes has the largest classification overhead. We also compared two numbers of instances for instance-based learning (i.e. 500 and 1000) because they have different classification overhead due to the different size of database that the classifier looks up to determine the most similar previous case. Note that perceptron has remarkably low classification overhead compared with other machine learning algorithms. For perceptron, the classification process involves calculating the output from a multivariable linear equation consisting of the attributes weighted by the coefficients, which requires small amount of the computation. As a result, perceptron shows the lowest cost in terms of the classification time amongst three machine learning algorithms by showing less than 0.05ms of the classification time.

C. Scheduling Accuracy

We then investigated the scheduling accuracy of the three machine learning algorithms. For an unbiased evaluation, we turned off the adaptive online training mechanism, and the online training phase and runtime scheduling phase were uniformly switched in one period (5 of training and 15 of runtime scheduling). A total of 10 periods were repeated for the average. Also, we demonstrated the adaptability of MALMOS to various network conditions and computation capability of remote resources by comparing with two static scheduling policies: threshold-based and linear equation-based scheduling cases. For the threshold-based scheduler, we empirically observed the performance for offloading and local processing and picked up the input size where the performance (i.e. execution time) of offloading begins to be better than local processing or vice versa. For instance, if the matrix size is bigger than 170×170 , offloading matrix

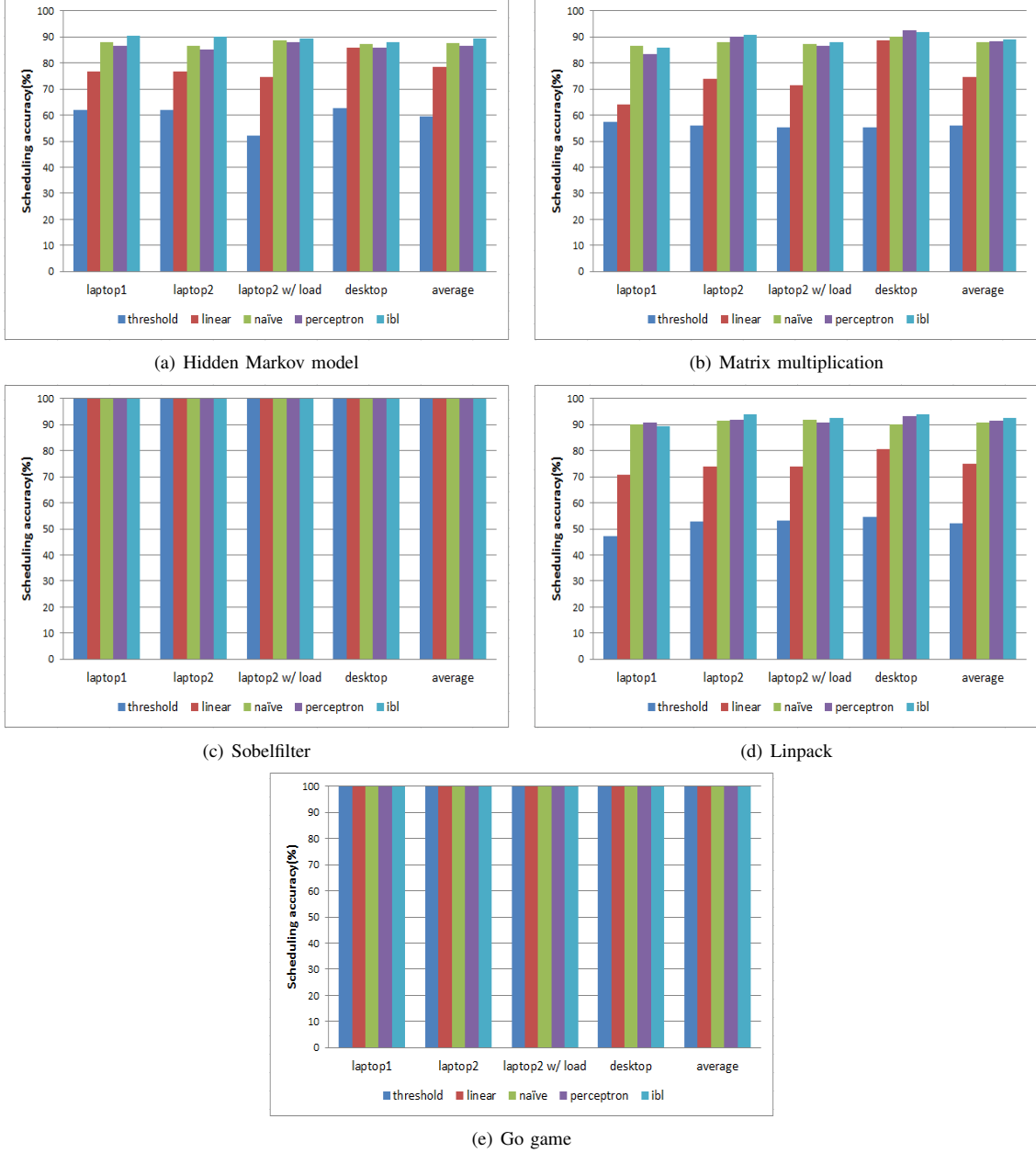


Figure 5. Scheduling accuracy for three machine learning algorithms and two static schedulers

multiplication to the workstation located in local area network shows better performance than local processing. As a result, we selected the data size corresponding to 170×170 of matrix size (i.e. 231KBytes) as the threshold so that when the data size is bigger than the threshold, the scheduler makes decision to offload the computation. Therefore, the threshold-based scheduler does not account for the network conditions. For the case of the linear equation-based scheduling policy, we established a two dimensional linear equation that distinguishes the cases where offloading or local pro-

cessing has better performance than the other. In order to define this linear equation, we gathered the performance of offloading each application to the workstation varying the network bandwidth. In this experimental setup, we measured the scheduling accuracy by offloading each application to four different remote servers, while varying the network bandwidth and the size of input. In order to change the network bandwidth, we wrote a simple script in which the TC command randomly switches among the three network bandwidth described in Table 1 in a certain interval during

the experiment.

As shown in Figure 5(a), (b), and (d), MALMOS with instance-based learning, perceptron, and naïve Bayes has better scheduling performance than the threshold-based and linear equation-based scheduling policies in the cases of hidden Markov model, matrix multiplication, and Linpack by showing averagely 86%~93% of the scheduling accuracy over four remote resources setups (for example, instance-based learning shows averagely 93% of the scheduling accuracy for Linpack in Figure 5(d)). The threshold-based scheduler considers only the size of input data as the decision factor, but does not reflect the dynamic network bandwidth. Consequently, the threshold-based scheduler shows the poorest scheduling performance in all experimental scenarios. Particularly, the linear equation-based scheduling policy has similar performance as three machine learning-based schedulers in the cases of offloading applications to the workstation, since the linear equation-based scheduler has been defined based on the offloading performance with the workstation while varying the network bandwidth. Nevertheless, the linear equation-based scheduler shows poor scheduling accuracy ranging from 64% (in case of matrix multiplication with first laptop setup) to 77% (in case of hidden Markov model with second laptop setup) in different remote server setups, such as laptop or laptop with CPU load. This is because other remote servers have different computing power capabilities, hence the offloading performance can be quite different with the case of offloading to the workstation.

On the other hand, Figure 5(c) and (e) show that all scheduling cases have 100% scheduling accuracy for sobelfilter and Go game in our experimental setup. As we mentioned earlier, sobelfilter and Go game are communication-intensive and computation-intensive applications, respectively. For this reason, sobelfilter always prefers local processing and Go game always prefers offloading. Even in these extremely simple cases, MALMOS demonstrates the scheduling performance as good as the threshold- and linear equation-based scheduler. It is worth noting that we established different threshold values and linear equations for each application. Even though we used three machine learning algorithms for each application without any change, MALMOS works consistently well in all of our experimental setups.

To evaluate end-to-end application performance, we measure the offloading time of Linpack. Table 2 shows total execution time of 150 consecutive Linpack runs with the threshold-based, linear equation-based, and MALMOS (instance-based learning). For this experiment, we used the threshold- and linear equation-based scheduling policies derived from the observation of the offloading performance with the workstation as the remote resource. With three scheduling policies, the mobile devices offloads Linpack to the laptop equipped with Intel Core2 Duo 2.0Ghz processor

Table II
TOTAL EXECUTION TIME OF 150 CONSECUTIVE LINPACK RUNS WITH
THREE SCHEDULING CASES: THRESHOLD, LINEAR EQUATION AND
INSTANCE-BASED LEARNING SCHEDULERS

	Threshold	Linear	IBL
Exec.time(s)	920.84	818.97	768.86

running CPU load by Stress, while changing the network bandwidth as described in the previous experiment, and we measured total execution time of 150 Linpack runs with different input data sizes. As shown, the threshold-based scheduler has the longest execution time due to the lowest scheduling accuracy. The linear equation-based scheduler also shows 6.5% longer execution time than MALMOS.

VI. CONCLUSION AND FUTURE WORK

In this paper, we propose a novel machine learning-based mobile offloading scheduler with online training, MALMOS. MALMOS is designed in a modular fashion to generally apply to any types of mobile offloading frameworks. Also, we propose an adaptive online training mechanism in which the online training phase and runtime scheduling phase are dynamically switched according to the scheduling accuracy. To evaluate our work, we applied MALMOS to DPartner, a Java-based on-demand offloading framework. Also, we utilized three different machine learning algorithms for the scheduling classifier and compared the overhead in terms of the training and classification time. Furthermore, we validated the ability of MALMOS to adapt to various network conditions and computing power capabilities of the remote computing resources by comparing the scheduling performance with threshold- and linear equation-based scheduling policies. In the evaluation, we observed that MALMOS has 10.9%~40.5% higher scheduling performance than two static scheduling policies under various network conditions, computing power capabilities of the remote servers, and application complexity.

For future work, we plan to apply our work to various types of mobile offloading frameworks with more machine learning algorithms and target real world applications. Also, we will consider scheduling scenarios where several remote resources are available simultaneously.

ACKNOWLEDGMENT

This material is based upon work supported in part by the National Science Foundation under Grant No. 1265341, 1234983 and 1339737. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

This work is partially funded by the National Natural Foundation of China (Grant No. 61361120097, 61222203 and 61300002).

REFERENCES

- [1] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Ch, and P. Bahl, "Maui: making smartphones last longer with code offload," in *In proceeding of International Conference on Mobile Systems, Applications and Services(MobiSys)*. ACM, 2010, pp. 49–62.
- [2] R. Kemp, N. Palmer, T. Kielmann, and H. E. Bal, "Cuckoo: A computation offloading framework for smartphones," in *In proceeding of International Conference on Mobile Computing, Applications and Services(MobiCASE)*, vol. 76. Springer, 2010, pp. 59–79.
- [3] J. Flinn, S. Park, and M. Satyanarayanan, "Balancing performance, energy, and quality in pervasive computing," in *In proceeding of International Conference on Distributed Computing Systems(ICDCS)*, 2002, pp. 217–226.
- [4] B. G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *In proceeding of the European Conference on Computer Systems(EuroSys)*, 2011, pp. 301–314.
- [5] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "Comet: code offload by migrating execution transparently," in *In proceeding of 10th USENIX conference on Operating Systems Design and Implementation(OSDI)*. ACM, 2012, pp. 93–106.
- [6] S. H. Hung, C. S. Shih, J. P. Shieh, C. P. Lee, and Y. H. Huang, "Executing mobile applications on the cloud: Framework and issues," *Computers and Mathematics with Applications*, vol. 63, no. 2, pp. 573–587, 2012.
- [7] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: enabling remote computing among intermittently connected mobile devices," in *In proceeding of ACM International Symposium on Mobile Ad Hoc Networking and Computing(MobiHoc)*. ACM, 2012, pp. 145–154.
- [8] M. A. Hassan and S. Chen, "Mobile mapreduce: Minimizing response time of computing intensive mobile applications," in *In proceeding of International Conference on Mobile Computing, Applications and Services(MobiCASE)*, vol. 95. Springer, 2011, pp. 41–59.
- [9] Y. Kwon and E. Tilevich, "Energy-efficient and fault-tolerant distributed mobile execution," in *In proceeding of International Conference on Distributed Computing Systems(ICDCS)*. IEEE, 2012, pp. 586–595.
- [10] H. Eom, P. S. Juste, R. Figueiredo, O. Tickoo, R. Illikkal, and R. Iyer, "Machine learning-based runtime scheduler for mobile offloading framework," in *In proceeding of International Conference on Utility and Cloud Computing(UCC)*. ACM/IEEE, 2013, pp. 17–25.
- [11] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, "Refactoring android java code for on-demand computation offloading," in *In proceeding of International Conference on Object Oriented Programming Systems Languages and Applications(OOPSLA)*. ACM, 2012, pp. 233–248.
- [12] S. Imai and C. A. Varela, "Light-weight adaptive task offloading from smartphone to nearby computational resources," in *In proceeding of Symposium on Research in Applied Computation(RACS)*. ACM, 2011, pp. 146–152.
- [13] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojevic, "Adaptive offloading inference for delivering applications in pervasive computing environments," in *In proceeding of International Conference on Pervasive Computing and Communications(PerCom)*, 2003, pp. 107–114.
- [14] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: A machine learning based approach," in *In proceeding of SIGPLAN Symposium on Principles and Practice of Parallel Programming(PPoPP)*. ACM, 2009, pp. 75–84.
- [15] J. L. Berral, I. nigo Goiri, R. Rou, F. Julià, J. Guitart, R. Gavaldà, and J. Torres, "Towards energy-aware scheduling in data centers using machine learning," in *In proceeding of International Conference on Energy-Efficient Computing and Networking(e-Energy)*. ACM, 2010, pp. 215–224.
- [16] H. Eom, P. Juste, R. Figueiredo, O. Tickoo, R. Illikkal, and R. Iyer, "Opencl-based remote offloading framework for trusted mobile cloud computing," in *In proceeding of International Conference on Parallel and Distributed Systems(ICPADS)*. IEEE, 2013, pp. 240–248.
- [17] R. Wolski, N. T. Spring, and J. Hayes, "The network weather service: A distributed resource performance forecasting service for metacomputing," *Future Generation Computer Systems*, vol. 15, no. 5-6, pp. 757–768, 1999.
- [18] "Stress project," <http://people.seas.harvard.edu/apw/stress/>.
- [19] "Linpack," <http://0xbenchmark.appspot.com/>.
- [20] "Go game," <https://code.google.com/p/dpartner/>.