Computing While Charging: Building a Distributed Computing Infrastructure Using Smartphones

Mustafa Y. Arslan [*]
UC Riverside
marslan@cs.ucr.edu

Indrajeet Singh
UC Riverside
singhi@cs.ucr.edu

Shailendra Singh
UC Riverside
singhs@cs.ucr.edu

Harsha V. Madhyastha
UC Riverside
harsha@cs.ucr.edu

Karthikeyan Sundaresan
NEC Labs America, Inc.
Princeton, NJ, USA
karthiks@nec-labs.com

Srikanth V. Krishnamurthy
UC Riverside
krish@cs.ucr.edu

ABSTRACT

Every night, a large number of idle smartphones are plugged into a power source for recharging the battery. Given the increasing computing capabilities of smartphones, these idle phones constitute a sizeable computing infrastructure. Therefore, for an enterprise which supplies its employees with smartphones, we argue that a computing infrastructure that leverages idle smartphones being charged overnight is an energy-efficient and cost-effective alternative to running tasks on traditional server infrastructure. While parallel execution and scheduling models exist for servers (e.g., MapReduce), smartphones present a unique set of technical challenges due to the heterogeneity in CPU clock speed, variability in network bandwidth, and lower availability compared to servers.

In this paper, we address many of these challenges to develop CWC—a distributed computing infrastructure using smartphones. Specifically, our contributions are: (i) we profile the charging behaviors of real phone owners to show the viability of our approach, (ii) we enable programmers to execute parallelizable tasks on smartphones with little effort, (iii) we develop a simple task migration model to resume interrupted task executions, and (iv) we implement and evaluate a prototype of CWC (with 18 Android smartphones) that employs an underlying novel scheduling algorithm to minimize the makespan of a set of tasks. Our extensive evaluations demonstrate that the performance of our approach makes our vision viable. Further, we explicitly evaluate the performance of CWC's scheduling component to demonstrate its efficacy compared to other possible approaches.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Wireless communication

[*]Mustafa Arslan is currently a research staff member at NEC Laboratories America Inc., Princeton, NJ USA.

Keywords

Smartphone, Distributed Computing, Scheduling

1. INTRODUCTION

Today, a number of organizations supply their employees with smartphones for various reasons [1]; a survey from 2011 [2] reports that 66% of surveyed organizations do so and many of these organizations have 75–100% of their employees using smartphones. For example, Novartis [3] (with 100,000 employees in 140 countries) handed out smartphones for its employees to manage emails, calendars, as well as information about health issues; Lowe's [4] did so for its employees to have real time access to key product information and to allow managers to handle administrative tasks.

In this paper, we argue that in such settings, an enterprise can harness the aggregate computing power of such smartphones, to construct a distributed computing infrastructure. Such an infrastructure could reduce both the capital and energy costs incurred by the enterprise. First, this could reduce the number of servers to be purchased for computing purposes. For example, Novartis awarded a contract of $2 million to IBM to build a data center for their computational tasks [5]. If they could exploit the smartphones handed out to their employees to run some portion of their workload, it is conceivable that the cost of their computing infrastructure could have been reduced. Due to recent advancements in embedded processor design, now a smartphone can replace a normal desktop or a server running a dual core processor for computation. According to Nvidia, their Quad Core CPU, Tegra 3, outperforms an Intel Core 2 Duo processor in number crunching [6]; for other workloads, one can expect the performance of the two CPUs to be comparable.

Our second motivation for the smartphone-based computing infrastructure is that the enterprise could benefit from significant energy savings by shutting down its servers by offloading tasks to smartphones. The power consumed by a commercial PC CPU such as the Intel Core 2 Duo is 26.8W [7] at peak load. In contrast, a smartphone CPU can be over 20x more power efficient, e.g., the Tegra 3 has a power consumption of 1.2W [7, 8]. Since their computing abilities are similar, it is conceivable that one can harness 20 times more computational power while consuming the same energy by replacing a single server node with a plurality of smartphones. In fact, to harness the energy efficiency of embedded processors, cloud service providers are already pushing towards ARM-based data centers [9].

The construction and management of such a distributed computing infrastructure using smartphones however, has a number of associated technical challenges. We seek to articulate these challenges and build an efficient framework towards making such a platform viable. In particular, the biggest obstacles to harnessing smartphones for computing are the phone's battery-life and bandwidth. If a smartphone is used for computing during periods of use by its owner, we run the risk of draining its battery and rendering the phone unusable. Further, today data usage on 3G carriers are typically capped, and thus, shipping large volumes of data using 3G is likely to be impractical. Thus, our vision is to use these smartphones for computing when they are being charged, especially at night. During these periods, the likelihood of active use of the phone by its owner will be low. Moreover, the phone will be static and, will likely have access to WiFi in the owner's home (today, 80% of the homes in the US have WiFi connectivity [10]); this will both reduce fluctuations in network bandwidth, and allow the transfer of data to/from the smartphones at no cost.

We name our framework CWC, which stands for computing while charging. To realize CWC, we envision the use of a single server, connected to the Internet, for scheduling jobs on the smartphones and collecting the outputs from the computations. The scheduling algorithms executed on the server are lightweight, and thus, a rudimentary low cost PC will suffice. Smartphones are only utilized for computation when being charged. If an owner disconnects the phone from the power outlet, the task is suspended, and migrated to a different phone that is connected to a power outlet. Towards building CWC, our contributions are as follows:

- **Profiling charging behaviors:** While an enterprise can possibly mandate that its employees charge their smartphones when they are not being actively used, we examine the typical charging behaviors of smartphone owners. Using an Android application that we develop, we gather charging statistics on the phones of 15 volunteers. Our results demonstrate that, on average, a typical user charges his phone for up to 8 hours every night.
- **Scheduling tasks on smartphones:** As a fundamental component of CWC, we design a scheduler that minimizes the makespan of completing the jobs at hand, taking into account both the CPU and bandwidth available for each smartphone. Since the optimal allocation of jobs across phones is NP-hard, we design a greedy algorithm for the allocation, and show via experiments that it outperforms other simple conceivable heuristics.
- **Migration of tasks across phones:** CWC executes tasks on smartphones only when they are being charged. Tasks are suspended if phones are unplugged during execution. We design and implement an approach to efficiently migrate such tasks to other phones that are plugged in.
- **Automation of task executions:** The typical means of running an application on smartphones is to have the phone's owner download, install, and run the application. However, we cannot rely on such human intervention to leverage smartphones for a computing infrastructure. We demonstrate how task executions on phones can be realized in a completely automated manner. Note that, while we recognize the potential privacy implications of running automated tasks on smartphones, we simply assume here that an enterprise would not run malicious tasks on its employees' smartphones. Improving the isolation of tasks implemented in typical smartphone operating systems is beyond the scope of our work.
- **Preserving user experience:** Blindly executing tasks for extended durations on a smartphone being charged, can prolong the time taken for the phone to fully charge. We show that intensive use of a phone's CPU can delay a full charge by 35%. We design and implement a CPU throttling mechanism, which ensures that task executions do not impact the charging times.
- **Implementation and experimentation:** Finally, to demonstrate the viability of CWC, we implement a prototype and conduct extensive experiments on a testbed of 18 Android phones. Specifically, we show the efficacy of the scheduling and task migration algorithms within CWC.

2. RELATED WORK

To the best of our knowledge, no prior study shares our vision of tapping into the computing power of smartphones. However, some efforts resemble certain aspects of CWC.

Smartphone testbeds and distributed computing platforms. Publicly-available smartphone testbeds have been proposed [11, 12] to enable smartphone OS and mobile applications research. CrowdLab [13] and Seattle [14] provide resources on volunteer devices. There also exist systems where users voluntarily contribute idle time on their PCs to computational tasks (e.g., [15]). In contrast, our vision is not for the smartphone infrastructure to be used for research and testing, but to enable energy and cost savings for real enterprises. Moreover, the issues that we address have not been considered by these efforts. In addition to these systems, flavors of MapReduce for smartphones have been implemented (e.g., [16]). However, such efforts do not address the issues of detecting idle phone usage and partitioning tasks across phones with diverse capabilities. They do envision using phones to offer a distributed computing service.

The system that is closest in spirit to CWC is Condor [17]. Condor can be used to queue and schedule jobs across a distributed set of desktop machines. These machines are either dedicated to running jobs on them or are operated by regular users for routine activities. In the latter case, Condor monitors whether user machines are idle and harnesses such idle CPU power to perform the computations required by jobs. It also preempts computations on these machines once the users continue their routine use (i.e., the machine is no more idle).

While the above features of Condor may be similar to CWC, the two have the following key differences:

- CWC tries to preserve the charging profile of smartphones via its CPU throttling technique. This is a challenge not addressed by Condor since desktop machines do not exhibit such a problem.
- Desktop machines mostly differ in terms of their CPU clock speed, memory (RAM) and disk space. In a cluster, these machines are connected via Ethernet switches and this typically results in uniform bandwidth across machines. Thus, systems such as Condor do not typically consider machine bandwidth in their scheduling decisions. In contrast, smartphones have highly variable wireless bandwidths (in addition to their variable CPU clock speed and RAM). This can lead to sub-optimal scheduling decisions if bandwidth is not taken into account (details in Section 3 and Section 5).

Participatory sensing. Recent studies such as [18], advocate the collective use of the sensing, storage, and processing capabilities of smartphones. With participatory sensing [19], users collect and analyze various types of sensor readings from smartphones. Unlike these efforts, a distinguishing aspect of CWC is that the data to be processed does not originate from the phones. In addition, CWC allows the execution of a variety of tasks unlike above, where typically a fixed task (sensing) is supported. Finally, CWC seeks to leverage compute resources on smartphones, rather than tapping human brain power [20].

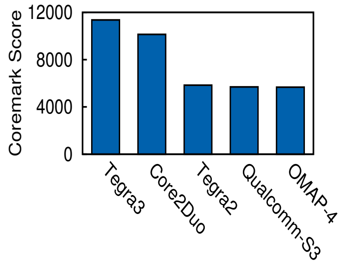


Figure 1: Benchmarking smartphone CPUs against the Intel Core 2 Duo.

Measurements of smartphones. There have been measurement studies [21, 22] to characterize typical network traffic and energy consumption on smartphones. In contrast, our focus is on developing a scalable platform for gathering measurements from the phones in CWC. Several prior studies [23, 24] have observed that phones are idle and are being charged for significant periods of time every day. We are however the first to recognize that these idle periods can be harnessed to build a distributed computing infrastructure.

Provisioning tasks on cloud services. Prior efforts have also tried to identify when the use of cloud services is appropriate (e.g., [25]), discuss the challenges involved in using them (e.g. [26]), or present solutions for provisioning applications (e.g., [27]). However, these efforts focus on traditional server-based cloud services. Prior efforts on managing resources in the cloud (e.g., [28]) do not tackle challenges associated with provisioning tasks on heterogenous resources nor deal with the variability of wireless links.

3. FEASIBILITY STUDY

In this section, we examine (a) the challenges associated with building a smartphone-based computing infrastructure and (b) the potential savings in capital and energy costs offered by such an infrastructure.

3.1 Challenges

Adequacy of computing power: The smartphone infrastructure is only attractive if it can effectively accomplish the computing tasks undertaken on today's servers. Due to rapid advances in embedded processor technologies, numerous smartphones with Quad Core CPUs are emerging [29]. Some of these CPUs offer clock speeds of up to 2.5 Ghz per core (Qualcomm snapdragon quad-core APQ8064) and their computational capabilities are *beyond that* of typically used server machines. To compare the performance of a smartphone CPU with that of typical desktop and server CPUs, we refer to the well known CoreMark benchmarks [30]. Figure 1 (borrowed from [8, 30]) shows the performance of major smartphone CPUs against a widely used desktop and server CPU—the Intel Core 2 Duo; the higher the CoreMark score, the better. We see that while the Nvidia Tegra-3 outperforms the Intel Core 2 Duo, the Core 2 Duo outperforms the other processors by more than 50%. This shows that state-of-the-art smartphones like Samsung Galaxy S-3 (running on Tegra-3 CPU) can only replace a single-core server or desktop machine. In our smartphone testbed, most of the smartphones are running on Tegra-2, Snapdragon S-3, and Ti OMAP-4 CPUs; in spite of this, we can execute a typical server job with two or three (of these older) smartphones.

Availability of idle task execution periods: Beyond the dramatic improvements in their compute capabilities, smartphones are attractive for a computing service because their resources are unused for long periods of time. Most users leave their phones idle overnight while the batteries are being recharged. When being recharged, a smartphone typically runs only light-weight background jobs (e.g., downloading e-mail) that require minimal computation and intermittent network access. Scheduling jobs on a phone during such periods is unlikely to impact the phone owners.

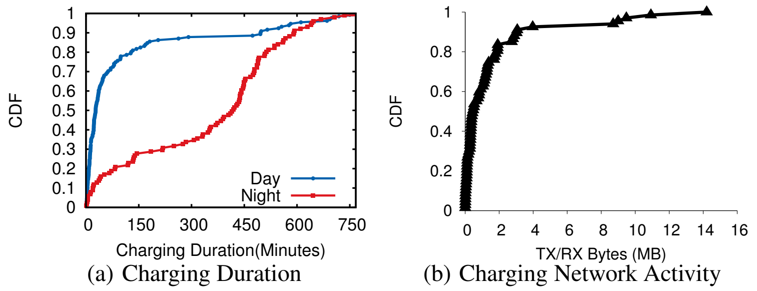
To identify and utilize idle periods, we have implemented an Android application to *profile* the charging behaviors of users (Apple iOS also supports the core functionality required). The application tracks three states on every phone: (a) *plugged*: when the user is charging the phone, (b) *unplugged*: when the phone is detached from the charger, and (c) *shutdown*: when the phone is powered off. When there is a change in state (i.e., *unplugged* to *plugged*), the application logs the change to a server along with a timestamp (of the user's local timezone). In addition, it logs the total bytes transmitted and received over all wireless interfaces (cellular and WiFi) when in the *plugged* state; this statistic is reset every time the phone newly enters the *plugged* state. The server parses the log files and computes for every charging interval of a particular user: (a) the duration of the interval, and (b) the number of bytes (transmitted and received) during that interval.

We conduct a study of realistic user behavior by having 15 volunteers (real users) install our application on their phones and gathering statistics. In Fig. 2(a), we plot the distribution of charging interval lengths, with every interval assigned to day or night; if the *plugged* state occurs between 10 p.m. and 5 a.m. of a user's local time, that interval is considered to be in the night, else in the day. We observe that the median charging interval is around 30 minutes and 7 hours long, at day and night respectively. In addition, there are fewer charging intervals in the night. This suggests that users charge their phones for an uninterrupted stretch of several hours at night. During the day, charging is interrupted frequently, resulting in a large number of short intervals.

We now focus only on the charging intervals at night. Fig. 2(b) plots the CDF of data transfers over all night charging intervals. Although the user is unlikely to be actively using the phone, there is background data in the form of periodic e-mail checks, push notifications from news and social media, etc.. However, we find that the total network activity is less than ≈ 2 MB for 80% of all night charging intervals.

Using the network activity data, we identify night charging intervals that can be considered idle to be the ones in which the data transfer is less than 2 MB. Fig. 2(c) shows that the users, on average, have at least 3 hours of idle charging at night. However, the characteristics highly depend on the individuals. Users with the highest idle durations (users 3, 4, and 8) have lower variability in their behavior; this suggests that they regularly charge their phones for 8 to 9 hours at night. In addition, users very rarely turn their phones off while charging (only 3% of the logs are in the *shutdown* state). The consistent low load at night (as also reported by [31]) suggests that idle usage patterns occur in large-scale settings as well. Given this, we speculate that this will provide an overlap of long idle charging times across users, yielding several operational hours for computing, without disturbing users' routine activities.

Next, we also examine the *plugged* and *unplugged* activity of each user. Our goal is to identify periods where, the phones are most likely to be unplugged. In our setting, we consider *unplugging* as a failure since we do not execute tasks when a phone is *unplugged*. Figure 3(a) plots the CDF of *unplugged* activity (failure) for all users. It is seen that the likelihood of failure between 12 AM to 8 AM is less than 30%. In CWC, we simply migrate such failed tasks to other phones that are still plugged in (discussed later).



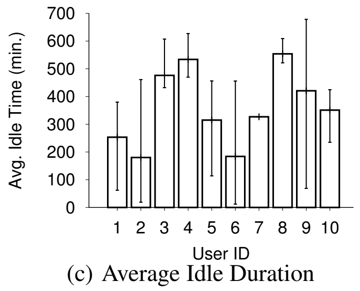


Figure 2: The median charging interval at night is around 7 hours and the data transfer is mostly below 2MB.

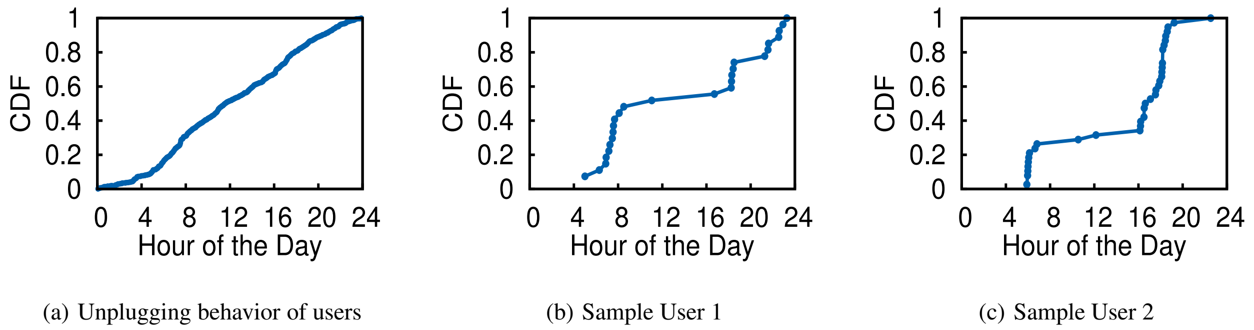


Figure 3: Availability of smartphones for CWC task scheduling.

Profiling an individual user's behavior can allow the prediction of device specific failures. This can help since tasks can be migrated to phones that are less likely to fail at the time of consideration. Figures 3(b) and 3(c) show the unplugging behaviors of two representative users from our study. The likelihoods of failure in both cases are very low between 12 A.M. and 6 A.M. It increases between 6 A.M. and 9 A.M. when people begin using their phones. During the day, the likelihood of *unplugged* activity is high; it decreases when phones are charged again at night.

Our study suggests that the charging behaviors of users are typically consistent at night, and offer an opportunity for harnessing the computing power of idle phones during these times.

Stability of the wireless network: To fully utilize the idle periods to execute jobs, a stable network connection is necessary. Since we only schedule jobs when a phone is on charge (typically at night), it is safe to assume that the channel qualities do not fluctuate much. The location of a device may, however, affect the bandwidth (due to fading); to account for temporally varying fading effects, a periodic (short) bandwidth measurement test is required prior to scheduling jobs on the phones. To examine the stability of these measurements over WiFi links, we conduct experiments at three different locations (within a 2 mile radius), when the phones are put on charge. Figure 4 depicts the results of such a bandwidth test for WiFi links where we run an *iperf* session from the phones to the server for 600 seconds.

We see that the variation in bandwidth for WiFi links is very low; this means that we can use infrequent (periodic) bandwidth measurements. Since we expect that communications between the smartphones and the supporting server will typically be via WiFi at users' homes, we conclude that bandwidth stability is not likely to be an issue. Cellular links can also be utilized as appropriate, but will require more frequent bandwidth measurements since they may exhibit high instability[32].

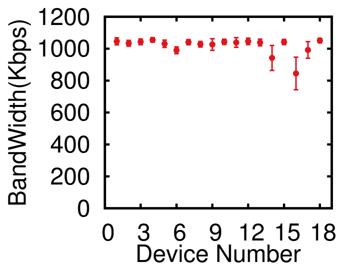


Figure 4: WiFi network stability.

Variability of bandwidth across smartphones: Although we showed that the bandwidth of a static smartphone is relatively stable, there may still be high variability in bandwidth *across* smartphones [1]. The task executable and the input data have to be shipped wirelessly to smartphones. This makes task completion times sensitive to the bandwidth variability across smartphones. To validate this in practice, we design a simple experiment where we have a central server (a regular PC) that interacts with 6 smartphones. The phones have identical CPU clock speeds but they differ in terms of their wireless bandwidths to the server. The server has 600 files to be processed by the phones (each phone finds the largest integer in the file). For each file, the typical cycle is the following. The server sends the file to one of the idle phones, which then processes the file and returns the result back to the server. If there are no idle phones (i.e., all phones are busy receiving and processing some file), the file is queued. Since all the phones are initially idle, the server can copy the first 6 files in parallel without any queue-

[1]Note that this is in contrast to the typical setting where desktop machines are inter-connected via Ethernet.

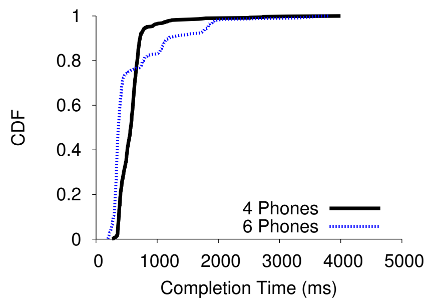


Figure 5: CDF of file processing times.

ing. The server logs the turn-around time for each file, which is computed as the difference between the time that the phone returns the result and the time that the file was queued. After this first experiment, we remove the two phones that have the "slowest" connections and schedule the 600 files on the remaining 4 phones. We observe from Fig. 5 that with 6 phones, 90% of the tasks finish in less than 1200 milliseconds. On the other hand, choosing a reduced number of phones albeit with fast wireless connections, improves the 90th percentile to 700 milliseconds (though the queueing delay increases). Our experiment reveals that simply accounting for the CPU clock speed and using all the phones results in poor task completion times. If this experiment were conducted on a cluster of 6 PCs with identical CPU clock speed, using more machines would have reduced the completion times (since the PCs would have the same bandwidth). In summary, one should also take wireless bandwidth into account when scheduling tasks across smartphones. This factor is unique to a smartphone environment and is not accounted for in systems such as Condor [17].

3.2 Benefits

Savings in infrastructure costs: Since the idle compute resources on already deployed smartphones are used, the cost borne by corporations to bootstrap the platform will be minimal in comparison to that in setting up a similar service on a server-based infrastructure. Companies have either to invest in buying hardware (e.g., servers, switches) or in outsourcing their tasks to third party cloud services. In addition, establishing computing infrastructure requires careful planning with regards to factors such as space, federal and state regulations, and the provisioning of power and cooling support. In contrast, the use of a smartphone infrastructure obviates such considerations. To leverage existing smartphones as the elements of a utility computing service, an enterprise will need no more than a central, lightweight server to identify idle resources and allocate them to computational tasks.

Savings in energy costs: A primary concern of cloud service providers is the power consumption in their data centers. A typical data center server can consume 26.8 Watts (Intel Core 2 Duo) to 248 Watts (Intel Nehalem) [33] of power, depending on the configuration. More importantly, this does not account for the power required for cooling. To calculate the total power consumption, we use an Average Power Usage Effectiveness (PUE) ratio [34] of 2.5; for every Watt consumed by a server, 2.5 watts are in addition consumed for cooling and power distribution. Extrapolating this, we can project the energy cost of a Intel Core 2 Duo server to be:

$$\frac{67}{1000}\text{KWH} \times 24 \text{ hrs} \times 365 \text{ days} \times \$0.127 = \$74.5/\text{year}$$

(using the average commercial price of 12.7c/KWH in the US in April 2011). Note that a more powerful server (like the Intel Nehalem) may cost up to \$689 /year.

In comparison to a datacenter server, the power consumption of a smartphone is as low as 1.2 Watts at peak load. We estimate the cost of operating a smartphone (with a similar model) to be:

$$\frac{1.2}{1000}\text{KWH} \times 24 \text{ hrs} \times 365 \text{ days} \times \$0.127 = \$1.33/year$$

Note that the PUE ratio does not apply in case of smartphones since they do not require any cooling. The above analysis suggests that energy costs of operating the smartphone computing infrastructure are significantly lower (by an order of magnitude) than using typical datacenter servers.

Example applications: Next, we describe some example applications that are suitable for execution on CWC in a real enterprise setting. The first is an example taken from the Condor website [17]. A movie production company can render each scene in a movie, in parallel, using smartphones. A second example is where, a department store gathers the sales records from several locations. These records can be partitioned and shipped to phones to quantify what types of goods are sold the most. We believe Lowe's would be a typical example for this [4]. Lastly, the IT department in an enterprise can gather machine logs throughout the day and analyze them for certain types of failures at night.

4. DESIGN AND ARCHITECTURE

In this section, we describe the design of CWC. We first describe the parallel task (job) execution model for CWC [2], and then seek answers to the following. **(a)** How can we predict task execution times?, **(b)** How can we implement automated task execution on smartphones without requiring direct user interaction?, and **(c)** How can we preserve user experience while the tasks are being executed on the phones?

Task model: In CWC, a task is a program that performs a computation on an input file, such as counting the number of occurrences of a word in a text file. Similar to the model in MapReduce, a central server partitions a large input file into smaller pieces, transmits the input partitions (together with the executable that processes the input) to the smartphones in CWC. Upon receiving the executable and the corresponding input, the phones execute the task in parallel and return their results to the central server when they finish executing the task. The central server performs a logical aggregation of the returned results, depending on the task. For the word count example, the server can simply sum the number of occurrences reported by each phone (obtained by processing their respective input partitions) to compute the number of occurrences in the original input file. We call such tasks *breakable tasks* to reflect that in this class, a task does not exhibit dependencies across partitions of its input and hence, can be broken into an arbitrary number of concurrent pieces.

While the above model is suitable for parallel tasks in general, some tasks *cannot* be broken into smaller pieces on which computations can be performed followed by merging the results, to produce a logical outcome. We call such tasks *atomic tasks*; such a task (and its input) can only be executed on a single phone due to the dependencies in its input. An example of an atomic task is photo filtering (e.g., blurring a photo). A blur is typically obtained by computing a new pixel value based on the neighboring pixels. Since the blurred pixel value depends on its neighboring pixels, a blur on a photo cannot be obtained by breaking the photo into smaller pieces, blurring the pixels in each individual piece and merging the results. Although an atomic task cannot be parallelized, there are still concurrency benefits when many such tasks are executed in batches.

[2]We use the terms task and job interchangeably.

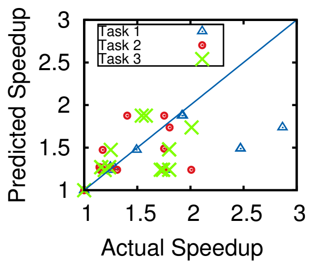


Figure 6: Predicted speedup vs. the measured speedup.

For example, if one needs to filter 1000 photos, each individual photo can be transferred to a phone and thus, multiple photos can be filtered in parallel. CWC accounts for both breakable tasks and batch atomic tasks in its scheduler (details in Section 5).

We realize that the RAM on most phones is smaller (1-2 GB) than most desktop machines (4 GB). This constraint can easily be overcome by splitting a given job input data into smaller fragments so that each data partition fits in the smartphone memory. We believe 1 GB RAM per phone is enough to run most of the MapReduce style distributed jobs. Note here that the work in [35] reports that the median job input size for such jobs is less than 14 GB. One can easily partition such jobs across 15-20 phones and still schedule them using CWC. Next, we describe how CWC predicts task execution times.

4.1 Predicting Task Execution Times

When a task is scheduled on a phone, there are two important factors that affect the completion time of that task. First, it takes time to copy the executable (i.e., binary) and the input file partition to a phone. This depends on the achievable data rate on the link between the phone and the central server that copies the data. Second, the same task takes different times to complete on different phones (depending on the computational capabilities of the phone). While "computational capabilities" is broad and can include characteristics such as the speed of reading a file from the disk (e.g., the SD card on a phone) or the size and speed of the cache, we only focus on the CPU clock speed of a phone; a phone with a fast CPU (in GHz) should execute a given task in less time as compared to a phone with a slow CPU.

Next, we introduce some basic notation that we use in the subsequent discussions. b_i is the time that it takes to copy 1 KB of data from the central server to phone i. c_{ij} is the time it takes to execute task j on 1 KB of input data using phone i. E_j is the size (in KB) of the executable for task j and L_j is the size (in KB) of input data that task j needs to process. Given this notation, the completion time of task j, when it is scheduled to run on phone i, is equal to $E_j * b_i + L_j * (b_i + c_{ij})$. The first term accounts for the time that it takes to copy the executable to the phone and the second term accounts for the copying of the input data and executing the task on it. If phone i is assigned a piece of job j's input file, we denote this by l_{ij} and one can simply replace L_j with l_{ij} in the above formula to account for executing input partitions.

The estimation of the b_i values are via direct measurements in CWC (bandwidth tests described earlier). While we focus on describing how task execution times are estimated in the following paragraphs, we emphasize that the bandwidth to a smartphone is taken into account when making scheduling decisions.

The estimation of c_{ij} for each phone task pair has to be low-cost since many such combinations may exist. To estimate c_{ij} values, we resort to a scaling technique where we first execute each task j on 1 KB of its input using the slowest phone with a S MHz CPU speed. If the slowest phone takes T_s milliseconds to locally execute task j on a 1 KB input (excluding the associated 'executable and data' copying costs), a phone with 'A' MHz CPU speed is expected to complete the same task in $T_s * \frac{S}{A}$ milliseconds. This technique avoids the cost of profiling each phone-task pair and as we show in Figure 6, is a fairly accurate representation of actual task completion times. In plotting Figure 6, we first run a task on the slowest phone in our testbed (HTC G2 with 806 MHz CPU). We then run the same task on all the other phones (the relevant executable code and data are transferred a priori). Comparing the actual runtimes of each phone i (denoted t_i) to the run time of the slowest phone (denoted t_s), we have the measured speedup, $\frac{t_s}{t_i}$. We then compute the expected speedup based on CPU clock speeds. If a phone has a X MHz CPU, then the expected speedup with respect to the HTC G2 is equal to $\frac{X}{806}$. We do this comparison for three different tasks (described later in detail in Section 6). Figure 6 shows that the CPU scaling model captures the actual speedup for most of the points (the points are clustered around the $y = x$ line), with a few exceptions where the actual speedup is higher than what is predicted by the model (the rightmost points on the x-axis).

The above model is used by CWC's task scheduler (described in Section 5), which runs on the central server and periodically assigns partitions of tasks to a set of phones based on the predicted task completion time. The phones return their results along with the time it actually took to locally execute their last assigned task. The scheduler then updates its prediction for each phone (and task) based on the reported execution times and uses it for predicting the run time in the following scheduling period. With this, CWC accounts for the few cases that the initial prediction fails to capture with regards to task execution.

4.2 Automating Task Execution

One of the key requirements of CWC is that a task be executed without requiring user input. The typical means of "running a task" on smartphones today is running an application (i.e., "app"). When a user wants to execute a new task on her phone, she needs to download and install the app. This process typically requires human input for various reasons (e.g., Android users are presented a list of app permissions and have to manually validate the installation). Such a mechanism is clearly not apt for CWC, since the tasks are to be dynamically scheduled on smartphones.

To run tasks on the phones, we leverage a cross-platform mechanism that uses the Java Reflection API for the Android OS. With reflection, a Java executable (i.e., a `.class` file) can dynamically load other executables, instantiate their objects and execute their methods, at runtime. This allows CWC to ship different task executables and input files to a particular phone in an automated fashion. In addition, the reflection functionality can be implemented as an Android service [3] thus, bypassing the need for human input. Note that dynamic class loading is not specific to Android; such capabilities are also available with other smartphone OSs (e.g., iOS permits this via shared libraries).

With reflection implemented on the smartphone side, CWC does not require any additional infrastructure at the central server. In fact, developers can continue to use their traditional Java programs and have them scheduled for parallel execution by CWC. Since Android can execute Java code, we just require developers to implement their tasks in Java (no knowledge of Android API required). In Fig. 7, we depict the flow chart of a typical CWC task. The `.java` source files are compiled into `.class` files at the cen-

[3]Android services do not display graphical elements to users and can run in the background.

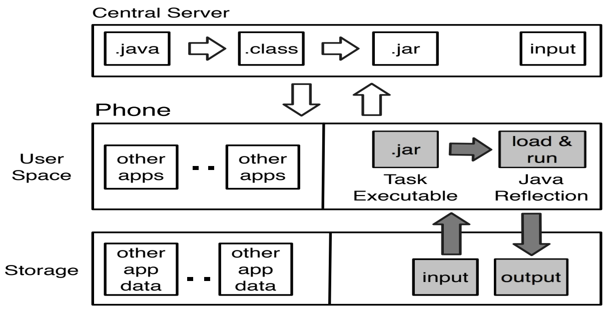


Figure 7: Flow chart of a CWC task (shown in shaded components).

```
public class Task {
    public static void main (String[] args) {
        executeTask(args[0]);
    }
    public static void executeTask (String filename
        ) {
    //read and process input
    }
}
```

Figure 8: Task.java to be compiled at the central server.

```
String path = getFilesDir().getAbsolutePath();
String jarFile = path + "/task.jar";
DexClassLoader classLoader = new DexClassLoader(
    jarFile, path, null, getClass().getClassLoader
    ());
Class[] types = {new String[]{}.getClass(),};
Class<?> myClass = classLoader.loadClass("Task");
Method m = myClass.getMethod("main", types);
Object[] passed = {new String[]{path + "/input.txt"
    }};
Object x = m.invoke(classLoader, passed);
```

Figure 9: Reflection functionality on the smartphone.

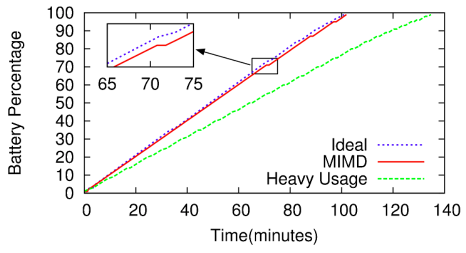


Figure 10: Charging times under different schemes for the HTC Sensation phones.

tral server, which are then packaged as `.jar` files using the Android tool chain (i.e., the *dx* command). The `.jar` file (containing the executable for the Android VM) together with the input data is copied to the phone. The phone extracts the `.jar` file and uses reflection to load and run the task, producing an output of results. Figure 8 shows the Java implementation of a typical CWC task. When the `.java` file is compiled and packaged in a `.jar` file, the task is executed on the phone using the code snippet in Figure 9. Thus, each phone concurrently processes the input partition (*input.txt*) assigned to it and this is transparent to the developers who implement their tasks (with the template in Figure 8) with a single machine in mind.

4.3 Preserving User Expectations

While predicting task execution times and automating them are important, we must note that phones are personal devices. Thus, first it is important to ensure that when a user chooses to use her phone, CWC stops the execution of the last assigned task to that phone so as not to adversely impact the end-user experience (e.g., task execution on the CPU can affect the responsiveness of the user interface). The tasks that are thus stopped, are then migrated to other phones that are still plugged in (as discussed).

Second, running tasks on phones that are are plugged in, should have a minimal impact on the charging times of the phones' batteries. We observe that a heavy utilization of a phone's CPU draws power and therefore, in some cases, prolongs the time taken to fully charge a phone's battery. Specifically, we conduct experiments where we first fully charge many types of phones (i.e., from 0% residual battery to 100%) in two settings; the first setting is without any job running on the phone, and the second setting corresponds to a case wherein a CPU intensive task is continuously run. As an example, we discuss results in the case of HTC Sensation phones, where we repeatedly run a CPU intensive task of counting the number of prime numbers in a large input file continuously during the charging period. We observe that while it takes around 100 minutes to complete full charging in the first setting, the time increases to 135 minutes in the second setting. Note that this increase could be phone-specific. In fact, we repeated the same experiment with HTC G2 phones and observed no significant effect (results are not reported due to space limitations). Our observation is that the more powerful the phone, the higher the penalty in terms of the increase in the charging period. Further note that, if the tasks are only scheduled after the phone is fully charged, there is no penalty (the phone remains fully charged); this is because the energy from the power outlet is directly applied to CPU computations. However, this would delay task processing and is thus avoided in CWC; moreover, users may not leave their phones plugged in until they are fully charged.

Our goal is to minimize the aforementioned adverse effect on a device's charging profile. If the CPU utilization can be controlled, we could achieve our goal. Prior approaches dynamically vary the voltage and/or the frequency of the CPU [36]. However, the modification of the voltage and frequency values require root privileges on the phone, and is therefore not applicable in our setting (using root privileges voids the phone warranty). Thus, our approach is to periodically pause the tasks being executed on the phones, and leave the CPU idle during such paused intervals. Next, we discuss when and for how long, we pause the execution of a task.

To begin with, our experiments demonstrate that the residual battery percentage (reported by the operating system) exhibits a predictable linear change with respect to time (as seen from Figure 10) in the case where no jobs run on the phone (referred to as the phone's charging profile). The rate of this linear change is specific to the device and the power source, but the relation remains linear across all the devices. When a task runs on the CPU of the phone, it draws power and thus, the charging profile deviates from this linear profile. We seek to minimize this deviation.

If there was no other task (e.g., background jobs not scheduled by CWC) running on the phone, we could determine the deviation due to CWC. Unfortunately, the process is complicated by the existence of such other tasks (possibly at different times) each of which with unpredictable CPU requirements and therefore power consumption patterns. Further, there could be fluctuations in the input power drawn from the supply (e.g. charging using a USB vs. a wall charger). Given this, our approach is to continuously monitor the rate at which the battery is charged, and either increase or decrease CPU utilization accordingly; the amount by which we increase the CPU utilization is called the scaling factor.

Specifically, we first measure the time it takes for the residual battery charge (δ) to increase by 1% of its preceding value, without any jobs running on the phone. This value of δ is referred to as the target charging parameter. Then, we run the task for a time period of $\delta/2$ and put the process to sleep for the next $\delta/2$ seconds. We repeat this process until the overall residual battery charge increases by 1 %. Let the time taken for this be $\beta(\geq \delta)$; this is referred to as the actual charging parameter. If $\beta = \delta$, there may be energy available to further ramp up the CPU utilization (the power from the outlet might be higher than what is required for charging). In this case, we decrease the sleep time during each δ period by a factor of 0.75, thereby inherently increasing CPU utilization; a new β is then computed based on the new settings. If $\beta > \delta$, the power drawn by the CPU is affecting the battery charging profile. Thus, we increase the sleep time by a factor of 2. Again, a new β value is computed. The process is repeated continuously. Note that the above strategy is akin a multiplicative increase/multiplicative decrease (MIMD) of the period for which the CPU is kept idle. Finally, since the phone's charging profile could change with time (as an example due to other tasks), we recompute the value of δ each time the residual battery charge changes by 5 %.

We plot the results with our adaptive MIMD based CPU scheduling in Fig. 10 for the HTC Sensation phones. The results from an ideal charging profile (no tasks) as well as a case where the CPU is heavily utilized without our approach are also shown. We see that our approach allows the phone to charge in a time that is almost the same as in the ideal case; the MIMD behavior of our approach is highlighted in the zoomed insert in the figure. Without our approach, the charging time increases by 35 %. Note here that, the use of the adaptive approach results in an increase in computation time of about 24.5 % compared to the heavily utilized scenario (due to the sleep cycles).

5. TASK SCHEDULING

In this section, we detail how tasks are scheduled in CWC. We are given a set J of jobs and a set P of smartphones. As discussed earlier, each job $j \in \mathcal{J}$ and phone $i \in \mathcal{P}$. The time it takes i to process x KB of j's input is given by

$$E_j * b_i + x * (b_i + c_{ij}) \tag{1}$$

where, E_j is the size (in KB) of job j's executable, b_i is the time (in milliseconds) that it takes phone i to receive 1 KB of data from the server, and c_{ij} is the time that it takes for phone i to execute the job j on 1 KB of input data. Our objective is to schedule the tasks across the phones such that the time it takes for the last phone to complete, T, (the *makespan*) is minimized. In the schedule, each job j's input can be split into pieces and each piece can be assigned to a phone. l_{ij} denotes the size (in KB) of job j's input partition assigned to phone i. $l_{ij} = 0$ simply indicates that phone i is not assigned any input partition of job j. u_{ij} is an indicator variable that denotes whether or not a partition of job j's input is scheduled to run on phone i. The scheduling problem (SCH) is then captured by the following quadratic integer program.

SCH: Minimize T

$$\begin{aligned}
\text{s.t.} \quad & \sum_j u_{ij} * (E_j * b_i + l_{ij} * (b_i + c_{ij})) \leq T, \ \forall i \in \mathcal{P} \\
& \sum_i l_{ij} \quad = \quad L_j, \ \forall j \in \mathcal{J} \\
& u_{ij} \in \{0,1\} \qquad \forall i \in \mathcal{P}, \forall j \in \mathcal{J} \\
& \sum_i u_{ij} = 1 \ \ \forall \text{ atomic } j \in \mathcal{J}
\end{aligned}$$

where we minimize the makespan, T. The first constraint requires that all phones finish executing their assigned tasks before T. The second constraint ensures that for every job, all of its input is processed. The last constraint ensures that atomic jobs are allocated to a single phone [4]. SCH reflects the general case for the minimum makespan scheduling (MMS) problem, which is known to be NP-hard. MMS is defined as: "*Given a set of jobs and a set of identical machines, assign the jobs to the machines such that the makespan is minimized*" [37]. A more general version of MMS is scheduling using unrelated machines (U-MMS), where each machine has different capabilities and thus, can execute tasks in different times. In both of these problems, only atomic jobs are considered. In other words, the goal is to assign each job to exactly one of the machines such that the makespan is minimized. SCH is a general case of U-MMS. We consider both atomic and breakable tasks and the machine capabilities are different. Since the special case of SCH (U-MMS) is NP-hard, the hardness carries over to SCH as well.

Our Solution: We address the SCH problem by solving the complementary bin packing problem (CBP), similar to the approach in [38]. In CBP, the objective is to pack items using at most $\|\mathcal{P}\|$ bins (with capacity C) such that the maximum height across bins is minimized. Here, the items correspond to the tasks and the bins correspond to the phones. The correlation between CBP and SCH can be drawn as follows. Let us assume that there is an optimal solution to CBP where the maximum height across the bins is M. If one rotates each bin 90° to the right, each bin visually appears as a phone in makespan scheduling. Items packed on top of each other in a bin correspond to input partitions assigned to a phone one after the other. Clearly, M corresponds to the maximum completion time across the set of phones in the rotated visualization. Thus, packing all items (tasks) using at most $\|\mathcal{P}\|$ bins (phones) and minimizing the maximum height across bins will minimize the makespan [5].

The pseudocode of our greedy algorithm to solve CBP is given in Algorithm 1. The idea is to first sort the tasks in decreasing order of local execution time. The first item in the sorted list is the one where $R_j * c_{sj}$ is the largest; s is the slowest CPU phone in the system and R_j is the remaining input size (in KB) of item (job) j that is yet to be assigned to some phone. Initially $R_j = L_j$.

In each iteration, we search for the first item in the list that can be packed in any of the previously opened bins (an open bin represents a phone that has previously been assigned some input partition). Note that determining whether an item can be packed in a bin depends on whether the current height of the bin plus the execution cost of that item in the particular bin is less than the bin capacity. If we can find such an item, we pack it in the bin with the minimum height at that time (i.e., the phone with the least total execution time). When packing such an item (line 6), we pack its largest input partition that can fit. If the item can fit without partitioning it, we prefer packing it as a whole.

[4]Although not currently addressed in the paper, CWC can handle memory constraints. One can add $l_{ij} \leq r_i$ to ensure that any job j's partition assigned to phone i is less than or equal to the phone's RAM (denoted here by r_i).

[5]The reader can refer to [38] for details.

Algorithm 1 Greedy Packing Algorithm

```
L : sorted list in decreasing order of execution time
C : bin capacity
repeat
    find the first item in L that can fit in any opened bin
    if such an item exists then
        pack the item in the bin with min. height
        if the item was packed as a whole then
            remove it from L
        else
            insert its remaining input in L
            re-sort L
        end if
    else
        if there are un-opened bins then
            open the best bin for the largest item in L
            pack the item in the opened bin
            if the item was packed as a whole then
                remove it from L
            else
                insert its remaining input in L
                re-sort L
            end if
        else
            cannot open any more bins
            cannot finish packing with C
        end if
    end if
until all jobs are packed
```

The idea behind our design is the following. If a task is broken down to N pieces, the central server would have to aggregate N partial results, which would be an extra overhead at the server when the phones return their results. Thus, if two packings produce the same minimum bin height, we would prefer one with fewer partitions. If packing an item as a whole is not possible (simply because doing so would result in the bin height exceeding the capacity), we pack that item's largest partition that can fit without violating the bin capacity (with the purpose of keeping the number of partitions low). If no item can fit in the opened bins (line 13), we check if we can open a new bin. If not, the algorithm cannot find a feasible packing for the given capacity. If we can open a new bin, we open the bin that would accept the largest item in L, with the minimum increase in its height (line 15). Clearly, such a bin is the one that minimizes Equation 1 for the largest item in L. After opening the bin, we again try to pack the item as a whole (line 17). If not, we pack the item's largest partition subject to the bin capacity.

The above algorithm is repeated multiple times for different selections of bin capacities. Here, we adopt an approach similar to binary search. We first determine an upper bound (UB) on the bin height. Clearly, the maximum bin height occurs when all items are assigned to the bin that maximizes Equation 1 (i.e, the worst bin). For the lower bound (LB), we initially pick a loose bound where all items are packed in a single magical bin that has the aggregate processing capability and the aggregate bandwidth of all bins; there are no other bins. This magical bin represents the ideal case where the inputs are partitioned without the executable cost. After determining these initial bounds, we execute Algorithm 1 with $C = \frac{(LB+UB)}{2}$. If the algorithm succeeds packing all items with bin capacity C, we let $UB = C$. If the algorithm cannot find a feasible packing with the initial C, we let $LB = C$. The algorithm is then repeatedly executed with $C = \frac{(LB+UB)}{2}$, until the algorithm succeeds with the minimum C. Here, the binary search simply reduces the search space for the minimum bin capacity, with which the algorithm packs all the items.

When CWC determines the schedule as described above, it starts copying the relevant executables and the input partitions to each phone. This is done on a per-partition basis; the next assigned task to the phone is copied only after the phone completes executing its last assigned task. When the phones inform the central server about a task completion, they report the partial results together with the time it takes to locally execute the assigned task. As described in Section 4, CWC uses such execution reports to update its prediction on execution times of tasks. If the same task is assigned to the same phone in the future (albeit with a different input partition), CWC uses the updated prediction for scheduling.

Handling Failures: In CWC's task execution cycle, some phones may naturally fail while executing a given task. In our setting, the term failure can correspond to a variety of cases. For example, when a phone is plugged off the charger, we treat it as a failed node since continuing to execute a CPU-intensive task on it would drain the battery (a critical concern for CWC). In CWC, such failures are communicated back to the central server whenever possible (i.e., when the phone still has a network connection), and the execution can be resumed from the point where it failed (details of task state migration are in Section 6). We call these class of failures where the phone maintains a connection with the server "online failures". Other scenarios may include harder failures, in which the phone loses its connection to the server (e.g., wireless driver suddenly crashes or the network connection is dropped), and thus, cannot report its failed state back to the central server (the description of detecting such failures at the central server is again deferred to Section 6). We call this class of failures "offline failures".

Assume that at time instant A, we compute a schedule X. With X, each phone i has a set of tasks X_i that it will execute as time progresses. CWC starts copying the executable and input partitions in X_i to i one task at a time and waits for i to either report a completion or a failure. If no report is received for the last copied task, say $last_i$ (due to an offline failure), CWC marks i as failed and inserts $last_i$ and all the remaining tasks in X_i to a list F_A that contains all failed tasks after A. If i reports completion, CWC simply copies the next task in X_i and again waits for reports. If on the other hand, i reports failure for $last_i$, the report contains additional information: **(a)** how much of the input was processed by i by the failure instance, and **(b)** what was the intermediate (partial) result associated with the processing. CWC still inserts $last_i$ (and all that remains in X_i) in F_A, but now $last_i$ is inserted with only the part of the input not processed by i (and the intermediate results are saved). Now assume that we have a new schedule to be computed at time instant B. Some new tasks have entered the system at this point and are awaiting scheduling. Now, CWC computes a schedule for all such new tasks and F_A combined. The reason that we avoid immediate re-scheduling of tasks in F_A and wait until B is to account for the possibility that failed phones may re-enter the system after a short period of unavailability (e.g., the user plugs her phone to the charger after a few minutes or the connectivity is restored). Note that this is in contrast with typical MapReduce architectures, where failures may result in long periods of unavailability [39].

6. IMPLEMENTATION AND EVALUATION

Our testbed consists of 18 Android phones with varying network connectivity and CPU speeds. The network interfaces vary from WiFi (both 802.11a and 802.11g are considered) to EDGE, 3G and 4G. The CPU clock speeds vary from 806 MHz to 1.5 GHz. Each phone registers with a central server and reports its CPU clock speed. We measure b_i values (bandwidth to each smartphone i) with the *iperf* tool. The phones host the CWC software, which maintains a persistent TCP connection with the server and permits dynamic task execution as instructed by the scheduler. To maintain long-lived flows, we use the $SO\_KEEPALIVE$ option in the connection sockets as well as implement custom application layer keep-alive messages. The latter also serve as a means of detecting offline failures. If a phone fails to respond to a preset number of keep-alive requests from the server, it is marked as failed. In our implementation, the keep-alive message period is 30 seconds and the number of response failures tolerated, is 3.

Several techniques exist to migrate failed tasks and resume their state on a target machine; for example, the authors in [40] modify the Android virtual machine (VM) itself to migrate the state of execution but this requires changes to the original Android VM. To make it more user and developer friendly, we ported JavaGO – a Java program migration library [41] to the Android. JavaGO is based on a "source-code-level" transformation technique, where the JavaGo translator takes the user program as input and outputs the migratory Java code. The translated code can run on any Java interpreter and can be compiled by any Just-in-Time (JIT) compiler. JavaGo provides flexibility by allowing programmers to annotate their code using three added language constructs to the Java language, namely *go*, *undock* and *migratory*. The *go* statement specifies the IP address of the machine where the failed application will be resumed. The *undock* construct specifies the area to be migrated in the execution stack while *migratory* construct declares which methods are migratory so that only those methods are modified by the JavaGo compiler. In CWC, we translate the annotated java task files with JavaGo translator to produce the migratory `.jar` task file. In case of a failure, the state of a task is saved and transmitted to the central server (via the *go* construct). Our server records the transmitted state but does not itself resume the computation at that state. At the next scheduling instant, the server sends the recorded state of each failed task to a newly assigned phone, which then resumes the task. Details on migration can be found in [41].

The server is implemented as a multi-threaded Java NIO server. Non-blocking threads allow the server to concurrently copy data to a phone while reading the completion reports of other phones. We host the server on a small Amazon EC2 instance to show its lightweight implementation and economical viability. The small instance is the default configuration offered by EC2. It offers one virtual core with 1.7 GB of memory, which represent a machine that is far less capable than state-of-the-art workstations. Currently, Amazon charges 8 cents per hour for a small Linux instance. This clearly shows that the lightweight central server in CWC incurs very small cost for a typical enterprise (although the exact value may change over time).

Prototype Evaluation: In evaluating CWC, we use a variety of tasks. The first task involves counting the occurrences of prime numbers in an input file. The second task is to count the number of occurrences of a word in the input file and the third task is to blur the pixels in a photo. While we are able to directly use the desktop Java versions of the first two tasks, doing this was not possible for the photo blurring task. The challenge relates to the lack of compatibility between the graphics classes on the desktop Java virtual machine and the Dalvik virtual machine in Android. While the code works on JVM, the reflection class loader on Android complained about the part of the code that reads the pixels from the image file (in particular BufferedImage class does not exist in Android). To eliminate the phone's reading the pixels directly from the image, we do the following modification. We first pre-process the pictures to read the pixels (at the central server) and create text files that contain a pixel value in each of its lines. Each phone was able to process the text files as before. After this, the server re-creates each photo from the blurred pixels returned by each phone.

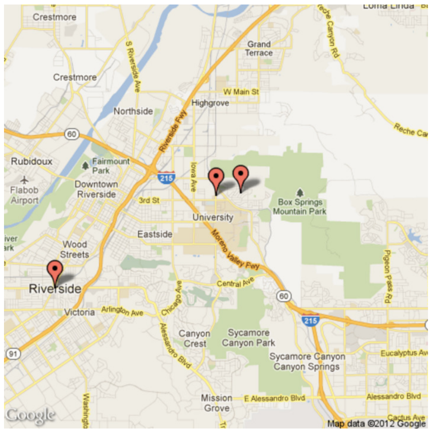
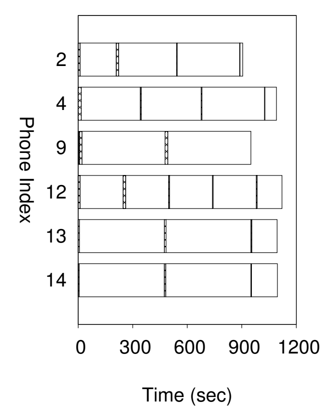


Figure 11: Map of the phone locations.

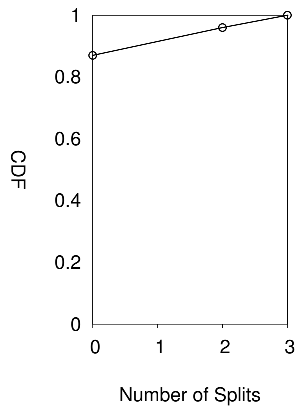
Comparison with simple practical schedulers: At the server, we also implement simpler alternatives to CWC. The first alternative splits each breakable job into $|\mathcal{P}|$ pieces without accounting for the different bandwidth and CPU speeds of phones in $\mathcal{P}$. The atomic jobs are assigned to phones in a round-robin manner (phone 1 gets atomic job 1 and so on). In the second alternative, both breakable and atomic jobs are assigned in a round-robin manner.

Setup: We distribute our 18 phones in three houses (the locations are shown in Fig. 11). In two of these houses, we have a 802.11g WiFi network and an abundance of interfering residential access points using the 2.4 GHz band. In the third house, we have a 802.11a WiFi AP without interference from neighboring APs. Of the 6 phones we place in each house (phones are plugged to power outlets), we associate 2 phones with the WiFi AP and 4 phones are configured to use varying cellular technologies (from the slowest EDGE to the fastest 4G). Before running the CWC scheduler, we initiate *iperf* sessions from each phone to the EC2 server and log the measured data rate in KBps (the inverse of this value is used as b_i). The workload comprises of the following. We have 50 instances of task 1 (counting primes) with varying input data sizes, we have 50 instances of task 2 (counting word occurrences) with varying input sizes and, 50 variable size photos to be blurred (atomic task 3).

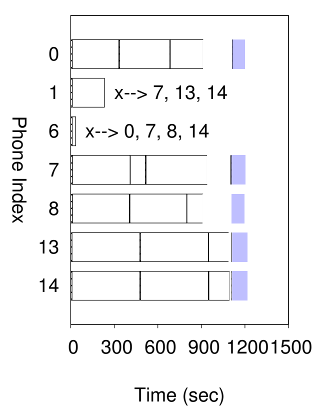
Results: In the first experiment, we run our greedy scheduler followed by the two alternate scheduling strategies described above; we do not consider phone failures. Fig.12(a) presents the task execution timeline for a select set of phones. We do not show the plots for every phone for better visualization (the patterns are similar across the phones). The vertical black stripes in a phone's timeline correspond to the time intervals where the phone is receiving the task executable and the corresponding input partition from the server. The white regions correspond to the time intervals where the phone executes the task locally. From Fig. 12(a), we observe that while some phones (2 and 9) finish their tasks earlier than others, the load is well balanced for most of the phones (4, 12, 13, 14 complete at similar time instants). Phones 2 and 9 finished early because of a mismatch between the expected speedup and the measured speedup (recall Fig. 6 in Section 5). In particular, these phones are faster than what is indicated by their CPU clock speeds and thus, finish earlier than the scheduler's prediction. We see that the difference in the completion times between the earliest phone (2 finishes at around 900 seconds) and the last phone (12 finishes at around 1100 seconds) is ≈20% of the makespan. In addition, our scheduler's predicted makespan of 1120 seconds was only 20 seconds more than the actual makespan of 1100 seconds. Fig.12(b) shows the CDF of the number of input partitions for each of the



(a) Timeline of CWC's task execution



(b) CDF of the number of input partitions



(c) CWC failure recovery timeline

Figure 12: Our greedy scheduler produces very few input partitions (b) and provides support for failure recovery (c).

150 tasks considered. An input partition of 0 indicates that the task was atomically assigned to a single phone. While 33% of the tasks by definition cannot be partitioned (the photo tasks are atomic), we observe that our scheduler preserves atomicity for most of the tasks (≈90%) and thus, significantly reduces the aggregation cost at the server. In contrast, we observe that the equal split strategy had a makespan of 1720 seconds, and produced a large number of input partitions since each breakable task is split into $|\mathcal{P}|$ pieces. While the round-robin strategy avoided excessive input partitions, it achieved a makespan of 1805 seconds. *In summary, our greedy scheduler is around* $1.6x$ *faster than the alternative schedulers and it achieves this while with almost negligible aggregation costs.*

In Fig. 12(c), we plot the timeline for a different run of the above experiment. Here, we introduce failures by unplugging three phones (phones 1, 6 and 17) at random instances during task execution (the plot again shows a subset of phones). As discussed in Section 5, in the next round of scheduling, our scheduler reschedules tasks from previously failed phones across the remaining set of phones. The x marks on Fig. 12(c) indicate the assignment of the failed tasks of a phone. The shaded task executions depict the execution of re-scheduled tasks. We observe that phone 1's tasks were partitioned across phones 7, 13 and 14. On the other hand, phone 6's failed tasks were re-scheduled across phones 0, 7, 8 and 14. Since phone 6 failed at the very beginning of its schedule, it had more tasks to be re-scheduled. Our scheduler mainly chose faster phones (phones 0, 7 and 8 completed ahead of time in the original schedule) to re-schedule these failed tasks. Overall, re-scheduling failed tasks required 113 seconds after the original makespan.

Benchmarking the Scheduler: Next, we try to get a lower bound on the makespan to benchmark the performance of our algorithm. This requires optimally solving the quadratic integer program formulated in Section 5, which is NP-hard owing to the integral nature. While the integral part can be relaxed by allowing the variables u_{ij} and l_{ij} to take fractional values and hence producing a loose lower bound, we still cannot use standard LP solvers to compute the bound owing to the quadratic nature. To address this, we can re-formulate the problem by transforming the first constraint to $\sum_j u_{ij} * (E_j * b_i) + l_{ij} * (b_i + c_{ij}) \leq T$, where now u_{ij} applies only to the first term. However, to prevent the solution from allocating jobs to a phone ($l_{ij} > 0$) without accounting for the shipping cost of its executable ($u_{ij} = 0$), we add another constraint $(1 - u_{ij})l_{ij} = 0, \ \forall i, j$. The latter can now easily be relaxed as $l_{ij} \leq L_j * u_{ij}$, thereby resulting in an LP relaxation, which can be solved to obtain a loose lower bound (smaller makespan than optimal due to relaxation) on the solution. Thus, we have $T_{relaxed} \leq T_{optimal} \leq T_{cwc}$, where T is the makespan produced by each of these solutions.

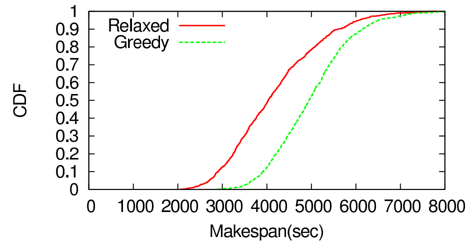


Figure 13: Comparison of the makespans of the greedy scheduler and the solution to the relaxed problem.

To understand how close we are to the optimal, we input various combinations of tasks and bandwidth profiles and solve the problem with **(a)** our greedy scheduler and **(b)** using the relaxed formulation above. We simulate the solutions by generating random b_i values between 1 and 70 milliseconds (the minimum and maximum values measured in our experiments). We consider the same set of 150 tasks and we get the c_{ij} values from the phones in our testbed. We generate 1000 random configurations and for each configuration, we first obtain the makespan for the relaxed formulation and subsequently we obtain the makespan produced by our greedy scheduler. Fig. 13 shows the CDF (over random configurations) of these makespans. It is seen that the median makespan of our greedy scheduler is approximately 18% worse than the relaxed formulation's solution.

7. CONCLUSIONS

In this paper, we envision building a distributed computing infrastructure using smartphones for the enterprise. Our vision is based on several compelling observations including (a) enterprises provide their employees with smartphones in many cases, (b) the phones are typically unused when being charged, and (c) such an infrastructure could potentially yield significant cost benefits to the enterprise. We articulate the technical challenges in building such an infrastructure. We address many of them to design CWC, a framework that supports such an infrastructure. We have a prototype implementation of CWC on a testbed of 18 Android phones. Using this implementation, we demonstrate both the viability and efficacy of various components within CWC.

Acknowledgements

We would like to thank Dr. Konstantina Papagiannaki, the anonymous reviewers and the program chairs Dr. K. K. Ramakrishnan and Dr. Patrick Thiran for their helpful suggestions towards improving the paper. This work was partially supported by the US Army Research Office MURI grant W911NF-07-1-0318.

8. REFERENCES

[1] Iphone in Business. `http://bit.ly/LE4dAp`.

[2] Enterprise Smartphone Usage Trends. `http://bit.ly/loIqE1`.

[3] Novartis: Apps for good health. `http://bit.ly/KEdJbh`.

[4] Lowe's : Building better customer service. `http://bit.ly/MiT9bk`.

[5] IBM gets US$2mn data center contract from Novartis. `bit.ly/L51P8i`.

[6] NVIDIA says Tegra 3 is a "PC-class CPU". `http://engt.co/srvibU`.

[7] S. Harizopoulos and S Papadimitriou. A Case for Micro-Cellstores: Energy-Efecient Data Management on Recycled Smartphones. In *DaMoN*, 2011.

[8] Variable SMP - A Multi-Core CPU Architecture for Low Power and High Performance. `bit.ly/n65KzQ`.

[9] Smart Phone Chips Calling for Data Centers. `bit.ly/LdM9fS`.

[10] WiFi Bandwidth Use in the U.S. Home Forecast to More Than Double in the Next Four Years. `http://yhoo.it/L9Po9A`.

[11] PhoneLab. `http://bit.ly/NYwRhI`.

[12] Almudeua Díaz Zayas and Pedro Merino Gómez. A testbed for energy profile characterization of IP services in smartphones over live networks. *Mob. Netw. Appl.*

[13] E. Cuervo, P. Gilbert, Bi Wu, and L.P. Cox. CrowdLab: An architecture for volunteer mobile testbeds. In *COMSNETS*, 2011.

[14] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Seattle: A Platform for Educational Cloud Computing. In *SIGCSE*, 2009.

[15] SETI@home. `http://setiathome.berkeley.edu`.

[16] P. R. Elespuru, S. Shakya, and S. Mishra. MapReduce System over Heterogeneous Mobile Devices. In *SEUS*, 2009.

[17] `http://research.cs.wisc.edu/condor/`.

[18] Tathagata Das, Prashanth Mohan, Venkata N. Padmanabhan, Ramachandran Ramjee, and Asankhaya Sharma. PRISM: platform for remote sensing using smartphones. In *ACM MobiSys*, 2010.

[19] D. Estrin. Participatory Sensing: Applications and Architecture. In *ACM MobiSys*, 2010.

[20] N. Eagle. txteagle: Mobile Crowdsourcing. In *Internationalization, Design and Global Development*, 2009.

[21] Earl Oliver. The challenges in large-scale smartphone user studies. In *ACM HotPlanet*, 2010.

[22] Hossein Falaki, Dimitrios Lymberopoulos, Ratul Mahajan, Srikanth Kandula, and Deborah Estrin. A first look at traffic on smartphones. In *IMC*, 2010.

[23] Alex Shye, Benjamin Scholbrock, Gokhan Memik, and Peter A. Dinda. Characterizing and modeling user activity on smartphones: Summary. In *ACM SIGMETRICS*, 2010.

[24] Earl Oliver. Diversity in smartphone energy consumption. In *ACM workshop on Wireless of the students, by the students, for the students*, 2010.

[25] Mohammad Hajjat, Xin Sun, Yu-Wei, Eric Sung, David Maltz, and Sanjay Rao. Cloudward Bound: Planning for Beneficial Migration of Enterprise Applications to the Cloud. In *ACM SIGCOMM*, 2010.

[26] Timothy Wood, Emmanuel Cecchet, K.K. Ramakrishnan, Prashant Shenoy, Jacobus van der Merwe, and Arun Venkataramani. Disaster Recovery as a Cloud Service: Economic Benefits & Deployment Challenges. In *USENIX HotCloud*, 2010.

[27] Azbayar Demberel, Jeff Chase, and Shivnath Babu. Reflective control for an elastic cloud application: an automated experiment workbench. In *USENIX HotCloud*, 2009.

[28] Thomas A. Henzinger, Anmol V. Singh, Vasu Singh, Thomas Wies, and Damien Zufferey. Static Scheduling in Clouds. In *USENIX HotCloud*, 2011.

[29] Quad-core smartphones: This is their year. `http://cnet.co/xvlHX5`.

[30] Coremark benchmark. `http://www.coremark.org/`.

[31] C. Peng, S. Lee, S. Lu, H. Luo, and H. Li. Traffic-Driven Power Saving in Operational 3G Cellular Networks. In *ACM MobiCom*, 2011.

[32] Justin Manweiler, Sharad Agarwal, Ming Zhang, Romit Roy Choudhury, and Paramvir Bahl. Switchboard: a matchmaking system for multiplayer mobile games. ACM MobiSys, 2011.

[33] Andrew Krioukov, Prashanth Mohan, Sara Alspaugh, Laura Keys, David Culler, and Randy Katz. NapSAC: Design and Implementation of a Power-Proportional Web Cluster. In *Workshop on Green Networking*, 2010.

[34] Green grid data center power efficiency metrics: PUE and DCIE. `bit.ly/MioRIt`.

[35] Antony Rowstron, Dushyanth Narayanan, Austin Donnelly, Greg O'Shea, and Andrew Douglas. Nobody ever got Þred for using Hadoop on a cluster. In *HotCDP*, 2012.

[36] Sebastian Herbert and Diana Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. ISLPED '07, 2007.

[37] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2004.

[38] JR. E. G. Coffman, M. R. Garey, and D. S. Johnson. An Application of Bin-Packing to Multiprocessor Scheduling. *SIAM Journal of Computing*, 7(1), Feb 1978.

[39] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX OSDI*, 2004.

[40] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. CloneCloud: elastic execution between mobile device and cloud. ACM EuroSys, 2011.

[41] Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. A Simple Extension of Java Language for Controllable Transparent Migration and Its Portable Implementation. COORDINATION, 1999.