Selective Checkpointing for Minimizing Recovery Energy and Efforts of Smartphone Apps

Li Li[1], Yunhao Bai[1], Xiaorui Wang[1], Mai Zheng[2], and Feng Qin[1]

[1]The Ohio State University, Columbus, OH, USA
[2]New Mexico State University, Las Cruces, NM, USA

***Abstract*—Unintended smartphone rebooting and unexpected shutdown (due to reasons like battery run outs, overheating, or automatic app upgrades) is annoying. What can be even worse is that a phone user has to restart, from the very beginning, the apps he or she was using when the phone got rebooted, because all the app states would be lost, especially when the number of apps in use is large. Hence, a recovery service is sorely needed for today's smartphones where apps are becoming increasingly complex. While checkpointing has long been used for desktop and laptop computers, such whole-system state preserving techniques cannot be applied to smartphones directly, due to the constraints of smartphones on energy, delay, and storage space.**

In this paper, we propose *SmartCP*, an intelligent checkpointing methodology, in order to reduce the energy required by a smartphone and the amount of efforts required by a user to recover the app states after the smartphone restarts. SmartCP *selectively* checkpoints the most important apps based on the apps' characteristics and predicted future usage, under the resource constraints of the phone. We propose a novel model that quantitatively analyzes the recovery energy and efforts of each category of smartphone apps and formulate selective checkpointing as a constrained optimization problem. We prototype SmartCP on Android and evaluate it using real-world traces as well as real user feedback. The results show that SmartCP outperforms two alternative app selection schemes by saving 28% more energy and 39% more recovery efforts on average.

I. Introduction

Due to the recent widespread use of smartphones, a variety of apps (such as Facebook and Amazon) have been developed. These apps have enabled smartphones to conduct different tasks that previously depend on a desktop/laptop in daily lives [1]. However, compared with desktops, smartphones are reported to have more unintended shutdown/reboot due to reasons like battery run outs, overheating, or automatic app upgrades [2], leading to a highly negative impact on the users' experience. What can be even worse is that a smartphone user usually has to restart, from the very beginning, the apps he or she was using when the smartphone got rebooted, because all the app states would be lost. This can be very frustrating.

It is important to note that although Android provides backup service for apps, this service can only be used to back up *persistent app data*, e.g., user configurations and preference, and cannot back up any runtime app states. App developers have to decide what data or states need to be persistent during runtime, demanding lots of expertise. Unfortunately, many app developers are inexperienced. For example, 26% of app developers have less than 2 years of developing experience [3]. Hence, despite backup services, app states are often insufficiently saved. As a result, upon a shutdown/reboot, the user has to repeat what he or she did before to recover all the app states, which requires non-trivial efforts from users and extra energy of smartphone. Therefore, a light-weight and developer-transparent recovery service is sorely needed for today's smartphones.

Application state preserving techniques such as hibernation have long been used on desktop/laptop computers. During hibernation, the states of all the applications (e.g., memory contents, open file descriptors, and thread-specific descriptors and signal masks) are saved to the hard drive before the machine is shut down. Once the machine is powered back on, everything can be resumed for the user to seamlessly continue the work. Unfortunately, such techniques have not yet been adopted for smartphones. The key reason is that phones are traditionally regarded as just a communication gadget (such as the feature phones ten to fifteen years ago). All the apps of those feature phones, like contacts and calendar, usually are very simple and do not have any states that must be preserved upon a reboot. Due to their much simpler designs, traditional feature phones also do not usually have many unintended reboots/shutdown (e.g., those caused by the software). However, as today's smartphones are becoming much more complex, in terms of both hardware and software, the demand for state recovery is significantly increasing.

It is not easy to apply existing whole-system state preserving techniques designed for desktops/laptops (such as checkpointing [4]) directly to smartphones, due to the more stringent constraints of smartphones on energy, running time, and storage space. According to the data reported by Yahoo [5], users have an average of 95 apps installed on their smartphones, 35 of which are used (on average) every day. Thus, multiple apps can often co-exist in the memory at the same time. If we directly adopt the whole-system strategy designed for desktop, i.e., checkpointing all the active apps, it would cause a high delay and consume a large amount of storage space as well as battery lifetime, leading to negative impacts on the user experience. Even with improved hardware of smartphone, naively saving the states of all the active apps is still unnecessary or suboptimal. This is because states of many active apps may be useless after reboot. Checkpointing those apps simply leads to the waste of energy, storage space

978-1-5386-3470-7/17/$31.00

and wear-out of flash memory. Thus, we propose a strategy called *selective checkpointing* that chooses only a subset of smartphone apps for checkpointing before an undesired reboot or shutdown. Such selective checkpointing introduces a key research challenge: Which apps should be selected for checkpointing? The limited resources should be given optimally to the apps whose states are more valuable (i.e., the apps that users would likely continue using after reboot and the ones that would take more energy/efforts to recover if not checkpointed).

In this paper, we propose SmartCP, a light-weight and developer-transparent selective checkpointing methodology that features 1) a novel cost model to estimate the recovery energy and efforts of each app, and 2) a constrained optimizer that minimizes the total estimated recovery energy and recovery efforts under given resource constraints. Based on collected real user traces, we analyze the characteristics of different smartphone apps. Two key observations are made to show that some apps have a high correlation between their recovery energy/efforts and activity depth (i.e., activity transitions required to reach the current state after launching the app), while other apps rely more on the number of user interactions. Based on such observations, we cluster apps into two different categories and propose a novel model that quantitatively analyzes the recovery energy and efforts of each category of apps. With the recovery model, we formulate selective checkpointing as a constrained optimization problem that minimizes the recovery energy and reduce corresponding efforts under given resource constraints. We implement a prototype of SmartCP on the Android platform and evaluate it with real user traces. The results show that SmartCP outperforms two alternative schemes by saving 28% more energy and 39% more recovery efforts on average. The overhead incurred by SmartCP is negligible. Specifically, this paper has the following major contributions:

- To the best of our knowledge, our work is the first effort that proposes selective checkpointing for smartphones.
- We propose a novel model that quantitatively analyzes the recovery energy and efforts of different categories of apps, which are clustered based on their usage characteristics.
- We formulate selective checkpointing as a constrained optimization problem that minimizes the recovery energy and reduces the efforts under given resource constraints.
- We prototype SmartCP and evaluate it with real-world smartphone user traces as well as real user feedback.

The rest of the paper is organized as follows. We present two key observations from app usage traces in Section II. Section III discusses the design and implementation of SmartCP. Section IV evaluates the performance of SmartCP with app usage traces and users' subjective test. We review the related work in Section V. Section VI concludes the paper.

II. Modeling Recovery Energy and Effort

Intuitively, we can estimate the "Recovery Energy/Effort of a running app" using the amount of energy that the app consumes before the reboot. However, users often exhibit different behaviors between recovering an app and during normal usage which requires different kinds of recovery efforts and different amounts of energy. In order to understand users' behavior on phone usage, we collected three weeks' traces from 11 users on their usage of 11 different smartphones with 5 different models. The traces contain 103 apps in 14 different categories. Moreover, collected traces include three key types of information relevant to user's behavior on phone usage: 1) activity [1] transition events when users interact with an app, 2) user input events, and 3) timestamps of user input events. Based on the traces, we have two key observations as follows:

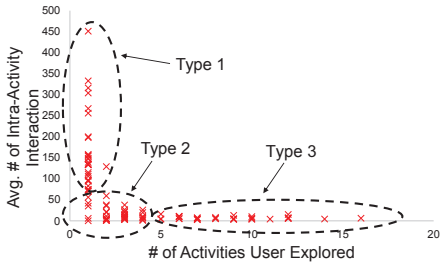


Fig. 1: Usage characteristics of 103 apps based on the number of explored activities and the average number of intra-activity interaction.

Observation 1: Different categories of apps require different amounts of energy and different kinds of efforts to recover the app states. From the collected usage traces, we observe that for apps such as Angry Birds [6], users only access one activity, but have a lot of intra-activity interactions (e.g., touching, clicking, and swiping). For other apps like Priceline [7], users usually explore multiple activities, while the number of intra-activity interactions is relatively small. In order to clearly understand the usage behavior, we represent the usage characteristic of each app with the following two features: 1) the number of activities a user explored, and 2) the average number of intra-activity interactions. Clustering analysis [8] is performed to group the apps into different clusters. In addition, Silhouette analysis [9] is adopted to determine the number of clusters. Figure 1 shows the clustering analysis results of the collected 103 apps. We can see that the apps can be mainly classified into three types: 1) low activity exploration, high intra-activity interaction; 2) low activity exploration, low intra-activity interaction; 3) high activity exploration, low intra-activity interaction. For simplicity, we group types 1 & 2 into one class, *Interaction-Intensive* apps, since both types have low activity exploration. Type 3 is classified into another class, *Activity-Intensive* apps. Different categories of apps require different kinds of effort to recover. For *Interaction-Intensive* apps (i.e., types 1&2), users often need to take the same number of touch actions to recover the app state. For *Activity-Intensive* apps (i.e., type 3), users need to follow certain activity transition sequence (hard for a user to remember) and may spend time waiting for the results from remote servers. Recovery energy is usually

[1] An activity is a core Android component, which is often represented as a full-screen window that users can interact with. Within an activity, typical GUI components are embedded that may transit the app from one activity to another with user input events.

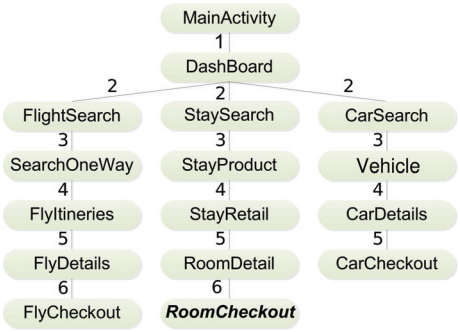


Fig. 2: An example of the activity tree. Each vertex represents an activity user explored, while each edge represents a transition from one activity to another.

highly correlated with the usage behavior. For *Interaction-Intensive* apps, the energy is mainly spent on processing user interactions, while the corresponding energy is mainly spent on retrieving data and rendering new activities for activity-intensive apps. Therefore, we differentiate these two categories of apps when estimating their recovery energy and corresponding recovery efforts.

Observation 2: Activity depth instead of the number of activity transitions is more accurate to estimate the recovery energy and efforts for type 3 apps. In this work, we define the activity depth of an app as the least number of activity transitions needed to restore the app state after relaunching. Figure 2 shows an example of the activity depth tree which is dynamically built as a user interacts with the app. It represents the structure of the activities a user explored. The vertexes represent the activities that a user has explored and indicate the corresponding activity depths. Each edge represents one activity transition. We can see that, in this example, if the smartphone reboots when the user is at the highlighted *RoomCheckout* activity (with a depth of 6), the minimum number of activity transitions required for recovery is 6. We observe that a user often switches back and forth between different activities when he/she interacts with an app. The activity depth fluctuates but the number of activity transitions keeps increasing. Thus, activity depth is more accurate to estimate the recovery effort and the corresponding recovery energy for type 3 apps.

Summary of Recovery Energy and Effort: Recovery energy and effort is highly correlated with usage behaviors. We model the recovery energy and effort of a running app in the following ways. Specifically, we estimate the recovery effort of an interaction-intensive (i.e., types 1&2) app R_{Inter} using the number of interactions and evaluate the corresponding energy required to process the interaction events. We estimate the recovery effort of an activity intensive (i.e., type 3) app $R_{Activity}$ using the activity depth and evaluate the corresponding energy required to process the activity transitions.

III. Design and Implementation

A. Design Overview

Figure 3 shows the architecture and work flow of *SmartCP*. The work flow mainly contains three phases.

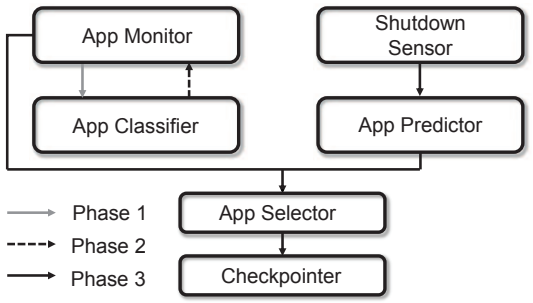


Fig. 3: SmartCP consists of six parts: 1) *App Monitor*, 2) *App Classifier*, 3) *Shutdown Sensor*, 4) *App Predictor*, 5) *App Selector*, 6) *Checkpointer*. The work flow mainly contains three phases.

Phase 1: One-time app classification. When a new app is installed, *App Monitor* records the usage dynamics (e.g., activity transition and user input events) as the user interacts with the app. *App Classifier* takes the information of the first n times usage as inputs and classifies the app into two categories (i.e., interaction-intensive or activity-intensive as described in Section II). Note that the classification is a one-time action for each newly installed app.

Phase 2: Touch-based information collection. After the app is classified, interaction or activity depth is monitored for interaction-intensive or activity-intensive apps, respectively.

Phase 3: On-demand *selective* checkpointing. When the smartphone is in use, once a shutdown event is detected by *Shutdown Sensor*, the remaining resources that can be used for checkpointing are first retrieved. *App Predictor* is then activated to predict the set of apps that might be reused after rebooting. Then, *App Selector* selects a set of apps to be checkpointed for minimizing the recovery energy and reduce the efforts a user would need to take under the given constraints. Finally, *Checkpointer* checkpoints the selected apps to the phone flash memory. After the checkpointing, the smartphone is allowed to reboot. After the phone is restarted, *Checkpointer* recovers the states of the selected apps, such that the user can resume what he was doing before the reboot.

The key design challenges of SmartCP are: 1) how to retrieve the set of apps that are most likely to be re-used after reboot, since the states of apps that users will not access in the near future are often useless; 2) How to select the right apps to be checkpointed to minimize the recovery energy and reduce the efforts with limited resource. Hence, in the following, we first introduce the design of App Predictor and App Selector. We then discuss other components and related design choices.

B. App Predictor

App Predictor predicts the set of apps that have a higher probability to be reused after the smartphone reboots. Given the highly limited resources on smartphones, we build a lightweight Naive Bayes classifier [10] for each app which is based on Bayes' theorem with independence assumptions between different features. While other machine learning techniques are available, we leverage the probabilistic models that require less computation. This is especially important because when a shutdown signal is detected there is limited time or

resource for computation. The model infers the probability of a target app (App_i) to be used again after reboot $P(App_i|C_i)$, given the context C_i and the prior probability $P(S_i)$. S_i is a binary variable to indicate whether app i will be used ($S_i = 1$) or not ($S_i = 0$).

$$P(App_i|C_i) = \frac{P(S_i = 1|C_i)}{P(S_i = 1|C_i) + P(S_i = 0|C_i)} \quad (1)$$

where

$$P(S_i|C_i) = P(S_i) \prod_{j=1}^{m} P(C_{i,j}|S_i) \quad (2)$$

m represents the number of adopted features. When a shutdown event is detected, *App Predictor* calculates the probability $P(App_1|C_1), P(App_2|C_2), ..., P(App_n|C_n)$ and selects the first n apps with the highest probability as the predicted results, where n is a parameter that can be configured by users.

Unlike previous work [11] that predicts the next app that a user is going to use, we predict a set of apps that will continue to be used within a certain time frame. In this work, we choose the following three context features ($C_{i,j}$): 1) time passed from the last access, 2) activity depth, 3) current time. These three features are chosen because they are the most indicative factors for the prediction [12]. Time intervals between two consecutive accesses of an app, activity depths and timestamps are dynamically recorded to update $P(S_i)$ and $P(C_{i,j}|S_i)$ as a user interacts with the app. The model is then constructed and updated on a smartphone during the usage process.

C. App Selector

App Selector contains three main steps: 1) determining the checkpointing cost, 2) retrieving resource constraints and 3) selecting *right* apps to be checkpointed.

CP Cost Analysis. This step calculates the checkpointing cost of a running app from three perspectives: storage, energy and time delay. More specifically, the storage cost is for saving the app's runtime states, mainly including 1) the contents of all memory regions, 2) a list of open file descriptors and the positions within the files they are pointing to, and 3) per-thread storage area descriptors and per-thread signal masks.

The time delay cost can be divided into two parts: interruption in the app execution (the checkpointing time) and the time for the checkpoint data to asynchronously reach the flash memory of the phone (the sync time). Furthermore, the checkpointing time is determined by two factors: the application's resident memory size at the specific point and the number of threads in the application [4]. The sync time only depends on the resident memory size. To dynamically retrieve the time required to checkpoint a certain app at runtime, we need to model the relationship among the time delay cost, application's resident memory and number of threads. To this end, we use a standard approach called *system identification* [13]. We infer their relationship by collecting data on real smartphones and establish a statistical model based on the measured data. The model $T = ax+by+1.349$ is used to represent the relationship among the time delay cost (T), the number of threads (x), and the application state size (y). Parameters a and b are determined through the collected data. The more threads an app contains and larger the app state size is, the more time it takes for checkpointing. Furthermore, the energy cost is proportional to the time needed for checkpointing an app.

When a certain event is detected, *App Selector* dynamically retrieves the resident memory size and the number of threads of a running app via the Linux proc file system [14]. Based on such information and the statistical model, *App Selector* calculates the storage cost, the energy cost, and the time delay cost of checkpointing each app in the to-be-reused set of apps.

Constraint Retrieving. This step checks the resource constraints, including storage, energy, and time delay, that can be used for checkpointing apps once a shutdown event is detected. Specifically, we access the environment variables to retrieve the currently available storage space. For retrieving the current battery level, we register a receiver to receive the *Intent.ACTION_BATTERY_CHANGED* broadcast message. Moreover, the difference among shutdown events is considered and the time constraints are calculated accordingly.

App Selection. We formulate the app selection problem as a constrained optimization problem. At a certain time point, the total energy/efforts required to recover all the apps to the states just before the shutdown/reboot is fixed. Thus, the more energy/efforts can be reduced through checkpointing, the less recovery energy is required and fewer recovery efforts the user needs to take after the reboot. We maximize the amount of recovery energy/efforts to be reduced through checkpointing under different resource constraints. As battery life is still an critical problem for smartphones, we first formulate the problem to maximize the reduction of recovery energy through checkpointing.

As discussed in Section II, the apps are divided into two categories including *interaction-intensive* apps and *activity-intensive* apps. Usually, a user interacts with apps in both categories and each category requires different amounts of recovery energy. To estimate the total recovery energy of the smartphone, we assign different weights for apps in different categories. This is because the energy required to process a touch event (e.g., interaction-intensive app) is usually much less than that of processing an activity transition which usually spends on retrieving data from remote servers and rendering it on the screen (e.g., activity-intensive apps). In this work, we use average processing energy of a user interaction event ($E_{Inter(i)}$) within an interaction-intensive app i and that of an activity transition ($E_{Activity(j)}$) within an activity-intensive app j as the weights of apps in different categories. Thus, the total recovery energy of all the running apps on a smartphone at a certain time point can be defined as:

$$E_{total} = \sum_{i=1}^{M} E_{Inter(i)} R_{Inter(i)} + \sum_{j=1}^{N} E_{Activity(j)} R_{Activity(j)} \quad (3)$$

M and *N* denote the numbers of *interaction-intensive* and *activity-intensive* apps, respectively.

Before we formulate the app selection process as an optimization problem, we first define the following notations: $E_{i/j}$ is the energy consumption for checkpointing app i/j. $D_{i/j}$ is the time required for checkpointing app i/j. $S_{i/j}$ is the storage

required to save the checkpoint image of app i/j. $P_{i/j}$ is the priority of each app that can be determined by users according to their preference. $\beta_{i/j}$ is a binary variable to indicate whether app i/j should be checkpointed ($\beta_{i/j} = 1$) or not ($\beta_{i/j} = 0$). The app selection problem can be formulated as shown in Equation 4.

$$\max\{\sum_{i=1}^{M} \beta_i P_i E_{Inter(i)} R_{Inter(i)} + \sum_{j=1}^{N} \beta_j P_j E_{Activity(j)} R_{Activity(j)}\} \tag{4}$$

Subject to one or more of the following constraints, depending on the shutdown scenarios to be discussed in Section III-D:

$$\sum_{i=1}^{M} \beta_i * E_i + \sum_{j=1}^{N} \beta_j * E_j <= E_{Threshold} \tag{5}$$

$$\sum_{i=1}^{M} \beta_i * D_i + \sum_{j=1}^{N} \beta_j * D_j <= D_{Threshold} \tag{6}$$

$$\sum_{i=1}^{M} \beta_i * S_i + \sum_{j=1}^{N} \beta_j * S_j <= S_{Threshold} \tag{7}$$

We can see that it is a 0-1 multidimensional knapsack problem (MDKP). In order to reduce the computing complexity of App Selector, we implement a heuristic algorithm [15] which combines Linear Programming with an efficient tabu search to approximately solve the multidimensional knapsack problem. The heuristic solution has polynomial time complexity and efficient approximation to the optimal. Moreover, the optimization only considers the *n* apps that have highest probability to be reused provided by App Predictor. The problem size is drastically reduced and the computation time is saved and bounded. The solution of the optimization problem will be stored in β_i and β_j where $1 \leq i \leq M$ and $1 \leq j \leq N$, which maximizes the recovery energy that can be reduced through checkpointing while still meeting the related resource constraints. The result will then be sent as the input to *Checkpointer*.

From the system perspective, SmartCP selects the apps to be checkpointed, in order to reduce recovery energy as represented in Equation 4. SmartCP can be also utilized to minimize the recovery efforts from users' perspective with different weights between the two categories of apps. In this work, we use the foreground time as the weights between two categories based on the observation that the more time a user spends on an app, the more important that app is for the user. Both the reduced recovery energy and recovery efforts will be evaluated in Section IV.

D. Other Components

App Monitor tracks the intra-activity interaction events (touching, clicking, swiping, etc.) and activity transitions of the running apps. Such events capture the main characteristics of the apps (i.e., interaction-intensive or activity-intensive). Furthermore, the timestamps of these events is also captured for assigning weights to apps in different categories.

TABLE I: Detect Different Shutdown Events

Event	How to detect
Low Battery	Instrument $shutdownIfNoPowerLocked()$ in the class $BatteryService$
Overheat	Catch the $BATTERY\_HEALTH\_OVERHEAT$ event
Watchdog	Instrument $run()$ in class $Watchdog$
Process crash	Catch the $SIGSEGV$ signal
Recovery mode requested	Instrument $bootCommand$ in class $RecoverySystem$

Whenever a user has interaction with an app, an input event will be generated and delivered to the *WindowManager* class in Android. Therefore, we implement *App Monitor* by retrieving the input events from WindowManager. Once an input event is detected, our code queries the *activity stack* to obtain the foreground activity and records the time stamp of that event. It is important to note that *App Monitor* only tracks basic usage information (e.g., touch, activity transition) and does not record any sensitive data. Thus, it does not generate privacy issue for app users.

App Classifier classifies apps into different categories according to their usage characteristics, including 1) Number of activities that the user has explored and 2) average number of intra-activity input events based on the observations as discussed in Section II. With the two features, we train a kNN (k Nearest Neighbors) [16] classifier using the collected usage information of the 103 apps (as shown in Figure 1) belonging to different categories. When a previously unknown app is installed and launched, *App Monitor* is invoked to track the two features of the app. The average of first n times usage information is then fed to the trained kNN classifier for classification. Training phase of the classifier is completed offline. After that, the model is used by different smartphones and the classification process is dynamically completed on the device. After the classification, interaction-intensive and activity-intensive apps can use intra-activity interaction and activity depth to estimate their recovery efforts, respectively.

Shutdown Sensor catches the unexpected shutdown/reboot events whenever the following scenarios occur. Table I lists the common events and the corresponding detection methods.

1) *Low Battery or Overheating*. When the battery level is critically low or its temperature gets higher than a certain threshold, the hardware sensors of the battery will raise an alarm signal to Linux kernel, which further notifies the *BatteryManager* class. Then *BatteryService* broadcasts a shutdown message to all the running services and apps on the smartphone. In this case, we have both the energy and time constraints for checkpointing.

2) *Reboot by WatchDog*. This happens when a deadlock in Android framework is detected or ANR (Application Not Responding) of a critical service occurs. In this case, we mainly use a time constraint for checkpointing.

3) *Process Crash*. Process crash happens when a fatal error, e.g., an invalid pointer, is thrown by the Linux kernel. If crash happens on the processes like *zygote* or *systemserver* that are indispensable to Android systems, the OS has to reboot. In this case, we mainly use a time constraint for checkpointing.

4) *Recovery mode is requested*. This scenario often occurs when the user installs a system update. Before rebooting

the phone for the update, some preparation task must be done. After the preparation, the *PowerManagerService* class is informed and it reboots the smartphone into the recovery mode. In this case, we mainly use a time constraint for checkpointing.

In addition, we may have an additional storage space constraint if the phone is currently out of space. Moreover, the checkpointing process can be effectively completed and the app states can be efficiently saved in these cases.

Checkpointer is to save the states (including the process, file system and the network state) of selected apps to the flash memory on the smartphone and automatically recover the apps' states after rebooting the phone. In this work, we implement our *Checkpointer* based on DMTCP (Distributed MultiThreaded Checkpointing) [4], a transparent user-level checkpointing package for distributed applications. Although our implementation is based on the version of DMTCP ported from Linux to the Android platform by K. Cheng [17], the proposed selective checkpointing methodology can be applied to other checkpointing tools such as CRIA (Checkpoint/Restore in Android) [18]. SmartCP is transparent to developers. The apps do not need any modification.

E. Discussion

In this section, we discuss some practical issues of implementing *SmartCP*.

How to handle the apps that have both high activity depth and high intra-activity interaction? To handle such apps, we can introduce one more category with these training data points being added. When a new app is installed, it will be classified with the revised kNN classifier. We do not include such a category in the current prototype because we do not observe any apps belonging to this category in our experiments.

Why not take checkpointing only on the server side? For apps like Amazon, it is true that certain states are already saved on the server side. However, the client-side (local) checkpointing is still highly desirable for the following reasons. First, not all the apps have the server side. Some apps only run locally on smartphones and do not have communication with remote servers. Second, users may not log into the app's server side until they get to a later stage, e.g., during the checkout activity. In this case, the apps' states rely on the client-side checkpointing to preserve.

How to guarantee the consistency between our client-side checkpoints and the server-side session states? If there is a session state maintained on the server side of an app, it requires coordination between our checkpointing mechanism and the server-side session process. One option is to implement a proxy service on smartphone that can automatically reconnect the app to the server side based on the recovered state of an app after rebooting the smartphone. To this end, we could borrow the ideas from previous work [19], and we leave this as future work.

IV. Evaluation

In this section, we first introduce the experimental methodology and baselines, and then discuss each experiment.

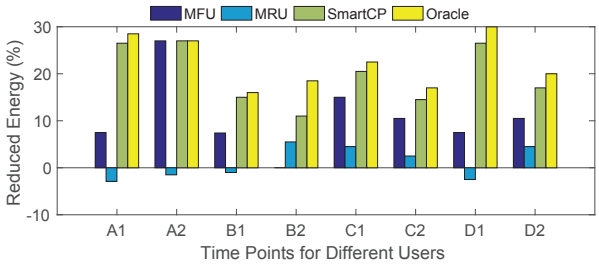


Fig. 4: Saving on Recovery Energy with Different Schemes.

A. Experimental Methodology and Comparative Schemes

We implement SmartCP on Android 4.4.4 and use a Monsoon Power Monitor [20] to measure the energy overhead of SmartCP. Two sets of real-world traces are adopted. First, we collect the smartphone usage data from eleven volunteer users for three weeks. The users have various backgrounds. Second, we adopt the user traces released by Rice University LiveLab [21], which include the traces of 34 users for a period of up to fourteen months. Moreover, subjective tests based on 20 real users are conducted. We focus on the evaluation of app selection instead of the checkpointing technique itself because our contribution is mainly on app selection and we use the same technique in [18] [4] for checkpointing after determining which apps to select.

Comparative Schemes. We evaluate SmartCP's effectiveness and efficiency by comparing it with three alternative schemes. The first one is *MFU* (Most Frequently Used), which checkpoints a number of apps that have highest access frequency among all the apps within the constraints. The second is *MRU* (Most Recently Used), which checkpoints a number of apps that have been most recently used among all the apps within the constraints. The third scheme is a theoretically-optimal but *unrealistic* scheme called *Oracle*, which makes the selection of apps for checkpointing based on the future usage (i.e., the usage after shutdown/reboot) of the apps. We measure the performance of each scheme using the *Reduced Recovery Energy* and *Reduced Recovery Effort* metrics, which is defined as the percentage of recovery energy/effort that can be reduced through checkpointing as discussed in Section III-C, the larger the better.

B. Reduced Recovery Energy with Different Schemes

In this section, we compare the reduced recovery energy. Comparison in all 11 users are conducted and Figure 4 shows the results from users A, B, C and D. We define the reduced energy as follows. E_{Manual_All} represents the energy consumption required for a user to manually recover the states of all the apps that he/she continues to use after reboot. Through checkpointing, states of the apps selected by different schemes can be saved and automatically restored after the reboot. $E_{Checkpoint}$ and $E_{Restore}$ represent the energy consumption of the checkpointing and restoring processes respectively. Due to the resource constraints, sometimes not all the apps that a user continues to use can be checkpointed. Thus, E_{Manual_N} represents the energy consumption required for users to manually recover states of the apps that are not checkpointed but continue to be used. Thus the percentage of energy saving of a certain scheme can be represented as

$\frac{(E_{Manual\_All}-E_{Manual\_N})-(E_{Checkpoint}+E_{Restore})}{E_{Manual\_All}}$. Intuitively, if more apps requiring high recovery energy are selected for checkpointing, less energy is required for manual recovery. Consequently, the reduced recovery energy is higher.

Figure 4 shows the reduced recovery energy under a 10s time constraint. For each trace, two time points are randomly selected to emulate the shutdown/reboot scenarios. We can see that SmartCP performs much better than MRU and MFU. This is because SmartCP selects the apps that require more recovery energy according to model as shown in Equation 4. The schemes MRU and MFU do not have this information and their performance is not stable. We can see that for points A1 and B2, the reduced energy of MRU and MFU is less than or equal to 0. This is mainly caused by two reasons. The first reason is that some apps selected by these schemes is in the initial state which requires little energy to recover, thus the checkpointing and restoring energy is larger than or equal to the energy required to manually recover those apps. The second reason is that users will not continue to use some of the apps after reboot. The difference between SmartCP and Oracle is mainly caused by the prediction error which is moderate.

C. Reduced Recovery Efforts with Different Storage Spaces

Recovery energy is important for smartphones, while recovery efforts are important for end users. In this section, we compare the reduced recovery efforts with different storage constraints. Comparison in all 11 users are conducted and Figure 5 shows the results from users E, F, G and H. For each trace, two time points are randomly selected (e.g., E1 and E2 for user A's trace) to emulate the shutdown/reboot scenarios.

Figure 5 shows that SmartCP performs much better than MFU and MRU under different storage constraints. For instance, at point E1 in Figure 5a, SmartCP reduces 40% of the recovery effort while MFU and MRU cannot effectively reduce any effort. At this point, seven apps exist in the memory. The recovery efforts of Half Brick, Amazon, Music are 0.59, 0.1, and 0.4 respectively. The recovery efforts of the other apps are negligible. In this case, SmartCP intelligently selects the Music app to checkpoint. Half Brick is not selected because the checkpointing cost exceeds the constraint. In addition, we can see that for some time points (e.g., F1, G2) the advantage of SmartCP under 100 MB storage constraint is higher than that under 50 MB storage constraint. This is because more apps exist in the memory at those points, when the constraint increases, it allows more space for optimization. When the storage space is large enough (e.g., 150 MB), most of the apps can be checkpointed. As a result, the difference between different schemes decreases.

D. Evaluation with Public Traces

In this section, we evaluate SmartCP using the traces from Rice University LiveLab [21]. In this trace, app usage information (such as the start time and duration of every app usage) of 34 volunteers was recorded for a period of up to fourteen months. We randomly select the usage traces of five users from the whole data set. Since Rice traces lack of the detailed information (e.g., activity and user interaction), we assign the activity depth and amount of interaction events to each app according to its usage characteristics. In each trace, we randomly select a time point as the point that the smartphone would reboot. Figure 6 shows the comparison of reduced recovery efforts under different schemes with 20-second time constraint for the five users. We can see that SmartCP always performs better than MFU and MRU for different users. This suggests that SmartCP is effective for apps with different usage characteristics.

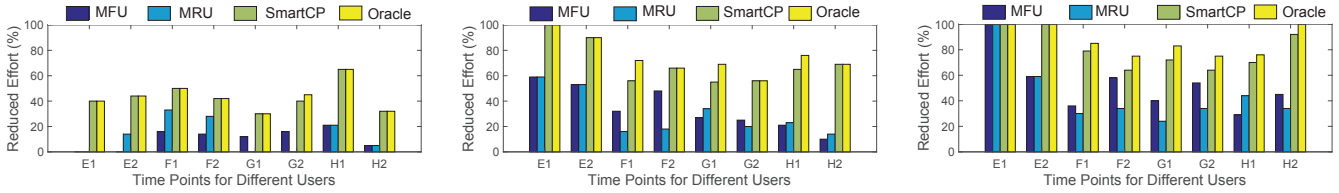
E. User-perceived Recovery Effort Reduction

Besides using traces, we have also evaluated the effectiveness of SmartCP's recovery effort model based on the feedback from real users. In this experiment, SmartCP is installed on the smartphones of 20 volunteers with different background such as researchers, students and engineers. During the daily usage, their smartphones are set to randomly reboot every several hours. After the reboot, users are asked to rank the apps they want to be checkpointed according to their preferences. In the meantime, SmartCP calculates the recovery efforts of the user-listed apps using Equation 4. Then, we generate two app sequences. The first sequence represents the users' preferences on the apps. The second one represents the apps ranked by SmartCP based on the reduced recovery efforts. Pearson Correlation [22] is then used to measure the correlation between the two sequences. The higher correlation between the two sequences is, the more effective SmartCP is in terms of modeling the recovery efforts perceived by real users. Figure 7a shows the Pearson Correlation Coefficient analysis among the reported cases. We can see that in 63% of the cases, the coefficient is larger than 0.7, indicating a strong positive correlation between the results from SmartCP and those from users' feedback. In 26% of the cases, the coefficient is between 0 and 0.7, which represent moderate positive linear relationship. Only 11% of the cases show negative relationship. This suggests that the recovery efforts modeled by SmartCP can effectively represent the real recovery efforts perceived by a wide range of users.

Figure 7b shows the overlap between apps selected by users and those selected by different schemes. At each shutdown/reboot point, users are required to generate the list of apps they need to checkpoint according to their importance. n is defined as the number of apps that can be checkpointed according to the provided app list under certain storage constraint. m is defined as the number of apps existing in both of the user-selected list and the list selected by certain scheme. m/n is then defined as the overlap ratio. We can see that the overlap ratio of SmartCP is much higher than that of other schemes in most cases. This suggests that SmartCP can intelligently select out the apps that users really need in most scenarios.

F. System Overhead

We measure the overhead of SmartCP on a Galaxy Nexus Phone. The memory overhead is 13.6MB which accounts for 0.45% of the whole smartphone. It is modest and remains stable during execution. The CPU utilization is less than 1%. The power consumption from SmartCP is about 3mW



(a) 50MB storage space. (b) 100MB storage space. (c) 150MB storage space.

Fig. 5: Comparison of reduced recovery efforts among different schemes for different users at various time points under three different storage space constraints.

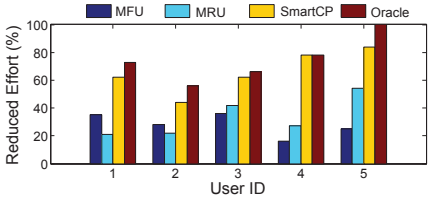
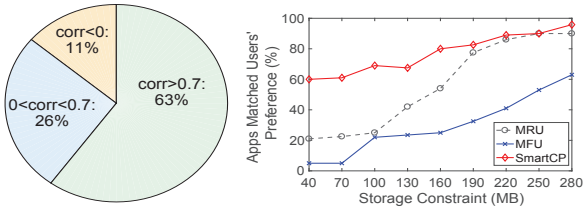


Fig. 6: Recovery effort reduction by different schemes under a 20-second time constraint, evaluated using traces from Rice University LiveLab [24].



(a) Correlation Analysis (b) Recovery Effort Reduction

Fig. 7: (a) Correlation between the recovery effort modeled by SmartCP and the recovery effort perceived by users. Corr represents the Pearson Correlation Coefficient between the two sequences. (b) Percentage of apps that matched user's preference among the maximum number of apps can be checkpointed with different schemes.

when the user does not have interaction which accounts for 0.194% of the whole smartphone and 27mW when the user interacts with a certain app which accounts for 1.04% of the whole smartphone. The time for SmartCP to complete the optimization and make the decision about which apps to checkpoint is less than 91 milliseconds. Overall, we can see that the overhead of SmartCP is modest.

V. Related Work

Checkpointing is widely adopted in computer systems for fault tolerance [23]–[25]. These generic checkpointing techniques cannot be directly applied to smartphones due to the limited resources available on smartphones. Moreover, generic checkpointing is unnecessarily expensive for smartphones since many apps are non-critical and thus may not need to be checkpointed at all. Checkpointing on Android is also studied [18]. For example, Hof et al. [18] extend the traditional checkpoint-restart mechanisms in a manner that leverages the characteristics of Android to save the core state of an app for migrating apps between devices. With this approach, they enable any app to become multi-surface. Thus, their purposes are also different from that of SmartCP.

VI. Conclusion

In this paper, we presented SmartCP, a selective checkpointing methodology that features a novel model to estimate the recovery effort of each active app, and a constraint optimizer that selects the most important apps for minimizing the estimated recovery efforts and energy under given resource constraints. We have built a prototype of SmartCP on Android and have evaluated it using real-world traces under various constraints. The results show that SmartCP has negligible overhead and outperforms two alternative schemes by saving 28% more recovery energy and 39% more recovery effort on average.

References

[1] D. Gilbert, "Microsoft wants to replace your PC with your smartphone ," www.ibtimes.co.uk.
[2] M. Barlett, "Android Random Restart," http://phonetipz.com.
[3] A. Cravens, "A demographic and business model analysis of today's app developer." http:// research.gigaom.com.
[4] J. Ansel *et al.*, "DMTCP: Transparent checkpointing for cluster computations and the desktop." in *IPDPS*, 2009.
[5] P. Sawers, "Android users app usage ," thenextweb.com.
[6] Rovio, "Angrybird. ," www.angrybirds.com.
[7] Priceline.com, "Priceline," http://www.priceline.com.
[8] H. S. Park *et al.*, "A simple and fast algorithm for k-medoids clustering." *Expert Systems with Applications*, vol. 36, pp. 3336–3341, 2009.
[9] P. J. Rousseeuw *et al.*, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis." *Journal of Computational and Applied Mathematics*, 1987.
[10] I. Rish *et al.*, "An empirical study of the naive bayes classifier." *IJCAI*, vol. 3, pp. 41–46, 2001.
[11] K. Huang *et al.*, "Predicting mobile application usage using contextual information." in *Ubicom*, 2012.
[12] C. Shin *et al.*, "Understanding and prediction of mobile application usage for smart phones." in *Ubicomp*, 2012.
[13] J. Sjoberg *et al.*, "Nonlinear black-box modeling in system identification: a unified overview." *Automatica*, pp. 1961–1724, 1995.
[14] E. Mouw, "Linux Kernel Procfs Guide," http://www.kernelnewbies.org.
[15] M. Vasquez *et al.*, "Improved results on the 0-1 multidimensional knapsack problem." *European Journal of Operational Research*, vol. 165, pp. 70–81, 2005.
[16] X. Wang, "A fast exact k-nearest neighbors algorithm for high dimensional search using k-means clustering and triangle inequality." in *IJCNN*, 2011.
[17] 0xlab , "Android DMTCP. ," http://github.com/0xlab.
[18] A. Hof *et al.*, "Flux: multi-surface computing in android." in *Eurosys*, 2015.
[19] F. Qin *et al.*, "Rx: Treating bugs as allergies-a safe method to survive software failures." in *SOSP*, 2005.
[20] Monsoon Solutions Inc., "Monsson Power Monitor ," www.msoon.com.
[21] C. Shepard *et al.*, "Livelab: measuring wireless networks and smartphone users in the field," in *SIGMETRICS*, 2011.
[22] J. Benesty, "Pearson correlation coefficient." *Noise reduction in speech processing*, vol. 20, pp. 1–4, 2009.
[23] M. Rahmanl *et al.*, "Checkpointing to minimize completion time for inter-dependent parallel processes on volunteer grids ." in *CCGrid*, 2016.
[24] Y. Li and Z. Lan, "Exploit failure prediction for adaptive fault-tolerance in cluster computing." in *CCGrid*, 2006.
[25] S. Di *et al.*, "Optimization of could task processing with checkpoint-restart mechanism." in *SC*, 2013.