CAreDroid: Adaptation Framework for Android Context-Aware Applications

Salma Elmalaki
University of California, Los Angeles
selmalaki@ucla.edu

Lucas Wanner
Department of Informatics and Statistics, Federal University of Santa Catarina, Brazil
lucas@lisha.ufsc.br

Mani Srivastava
University of California, Los Angeles
mbs@ucla.edu

ABSTRACT

Context-awareness is the ability of software systems to sense and adapt to their physical environment. Many contemporary mobile applications adapt to changing locations, connectivity states, available computational and energy resources, and proximity to other users and devices. Nevertheless, there is little systematic support for context-awareness in contemporary mobile operating systems. Because of this, application developers must build their own context-awareness adaptation engines, dealing directly with sensors and polluting application code with complex adaptation decisions.

In this paper, we introduce CAreDroid, which is a framework that is designed to decouple the application logic from the complex adaptation decisions in Android context-aware applications. In this framework, developers are required—only—to focus on the application logic by providing a list of methods that are sensitive to certain contexts along with the permissible operating ranges under those contexts. At run time, CAreDroid monitors the context of the physical environment and intercepts calls to sensitive methods, activating only the blocks of code that best fit the current physical context.

CAreDroid is implemented as part of the Android runtime system. By pushing context monitoring and adaptation into the runtime system, CAreDroid eases the development of context-aware applications and increases their efficiency. In particular, case study applications implemented using CAreDroid are shown to have: (1) at least half lines of code fewer and (2) at least 10× more efficient in execution time compared to equivalent context-aware applications that use only standard Android APIs.

Categories and Subject Descriptors

D.4.7 [**OPERATING SYSTEMS**]: Organization and Design—*Real-time systems and embedded systems*

MobiCom'15, September 7–11, 2015, Paris, France.

DOI: http://dx.doi.org/10.1145/2789168.2790108.

General Terms

Design, Performance

Keywords

Context-aware computing, Android, Context-adaptation

1. INTRODUCTION

Computation is becoming increasingly coupled with the physical world. It is now commonplace for mobile applications and systems to adapt their functionality to user location, connectivity status, device orientation, and remaining battery percentage. This ability for software to sense, react, and adapt to the physical environment has been termed *context-awareness* [31].

Context-aware applications typically feature multiple interchangeable methods and sets of parameters, each of which is activated when the system is under a specific set of physical conditions. A music streaming application, for example, may request lower quality streams from a server when using a cellular network radio than when using WiFi. Social network applications may discover and promote interaction between users in close physical proximity. A video encoding application may delay or lower the quality of its processing to save energy when the system is running out of battery.

When implementing context-aware applications, developers typically must probe sensors, derive a context from sensor information, and design an adaptation engine that activates different methods for different contexts. With adequate support from the runtime system, context monitoring could be performed efficiently in the background and adaptation could happen automatically [27]. Application developers would then only be required to implement methods tailored to different contexts. Just as file and socket abstractions help applications handle traditional input, output, and communication; a context-aware runtime system could help applications adapt according to user behavior and physical context.

In this paper we introduce CAreDroid, a framework for Android that makes context-aware applications *easier to develop* and *more efficient* by decoupling functionality, mapping, and monitoring and by integrating context adaptation into the runtime. In CAreDroid, context-aware methods are defined in application source code, the mapping of methods to context is defined in configuration files, and context-monitoring, and method replacement are performed by the runtime system.

Because applications using CAreDroid do not need to monitor and handle changes in context directly, they can be written using significantly fewer lines of code than would be required if only using the standard Android APIs. Because CAreDroid introduces context-monitoring at the system level, it can avoid the indirection overhead of reading sensor data in the application layer, therefore making context-aware applications more efficient.

To allow for transparent switching between polymorphic implementations—which are alternative implementations of the same method that either provide same functionality with different performances or provide alternative functionality for the same method—the CAreDroid framework is integrated as part of the Dalvik Virtual Machine (DVM). In particular, at runtime, CAreDroid intercepts the various sensor flows in order to determine the current context of the phone (where context parameters include energy, network connectivity, location, and user activity). CAreDroid uses this information along with the provided per-application configuration in order to dynamically and transparently trigger adaptations and to find the set of methods that, at any point in time, better suit the device's context.

1.1 Related Work

A context-aware system requires three major elements: (1) a set of mutually replaceable polymorphic methods, (2) a context monitoring system, and (3) an adaptation engine that switches between different methods based on the monitored context. We divide the related work based on the three elements mentioned above.

1.1.1 Developing Context-Dependent Behavior

We can identify three main strategies in developing the context-dependent behaviors as follows.

Code partitioning: Code partitioning for remote execution is based on the idea of cyber-foraging [30] where mobile devices offload some of the work to a remote machine with more resources like a server [23]. The server can then execute the heavy work on behalf of the mobile devices that have scarce energy. The idea of cyber foraging has been addressed in previous work with different aspects. Both Spectra and Chroma [16, 9] do program partitioning and run part of the code on a surrogate server. They both rely on an earlier work called Odyssey [26] that explored the idea of application adaptation based on network bandwidth and CPU load. Puppeteer [15] focusses on adaptation to limited bandwidth by making transcoding. Transcoding is a transformation of data to change the fidelity [26] of the application to save energy.

Reflective techniques: Reflection, originally noted by Smith [32], is a technique that has emerged in computing languages to provide inspection and adaptation of the underlying virtual machine. Reflective techniques have been exploited in mobile computing middleware to address context change. Reflective mechanisms have been used by Capra et al. [11] such that applications acquire information about the context, and then the middleware behavior and the underlined device configuration are tuned accordingly.

Alternate code paths: Alternate code paths or algorithmic choice has been addressed in energy-aware software literature such as Petabricks [5] and Eon [33]. The choice between alternate algorithmic implementations of the same code is done dynamically based on the energy availability. Each code path has different energy consumption in a tradeoff with quality or the accuracy of the result. A code path that is chosen by a pre-determined battery life has been explored in [21] in which tasks that have identical functionality are defined by developers. These tasks have different quality of service versus energy usage characteristics for embedded sensors application. In Green [7] a calibration phase is done at the beginning to determine the sampling rate—which eventually affects the accuracy of the result—in order to adapt to the available energy. Algorithmic choice has been further used in software libraries to deliver the best performance based on the hardware configuration [17, 22].

1.1.2 Context Monitoring

Efficient context monitoring has been studied throughly in literature. The work reported in [19, 24] provides frameworks for sensor-rich context monitoring. The main focus of that work is to minimize the energy consumption of the context-monitoring system. To further enhance the energy efficiency, Suman [25] exploits the temporal correlation between contexts in order to infer some context from others without reading the actual sensor measurements. To avoid performance degradation due to minimizing the energy consumption in context monitoring frameworks, the work in [20, 13] focuses on the optimization between continuous context monitoring, energy, latency and accuracy. Moreover, mobile operating systems currently support context monitoring functionalities. For example, the recently added *getMostProbableActivity()* API can be used to return the result of the Android OS activity recognition engine (e.g. biking or walking). Other examples are the Geofencing APIs provided by both Andorid and iOS which allow listening to the entrance and exit events from particular places and therefore allow for location based context applications.

1.1.3 Adaptation Engine

Prior work related to adaptation engines can be classified into two broad categories (i) *application-oriented adaptation engines* and (ii) *operation system-oriented adaptation engines*. The work reported in [10, 35] is a representative work for the first class. In this work, an application specific adaptation engines are designed and implemented with specific focus on energy-aware context adaptation. The work in [6, 34, 12] lies in the second category, where adaptation engines are proposed to perform context-aware OS functionalities such as context-aware memory management, context-aware scheduling, ... etc.

1.2 Paper Contribution

The work reported in this paper can be categorized under the class of *application-oriented adaptation engines*. While there is a rich body of work in designing application-specific adaptation engines, a systematic support for context-awareness is still missing from contemporary mobile operating systems. The work in this paper aims to fill this gap by providing OS support for the adaptation needed by context-aware Android applications.

In this paper we discuss the design and implementation of CAreDroid and present application case studies demonstrating the effectiveness of the system. Technically, we make the following contributions:

- We design CAreDroid, a framework for the implementation of context-aware polymorphic methods and for

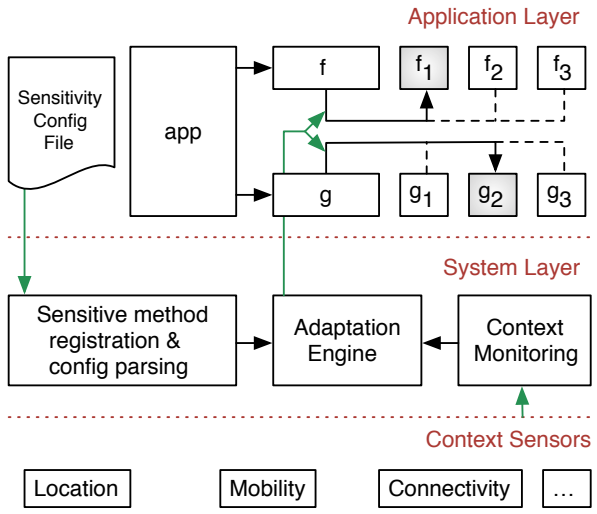


Figure 1: CAreDroid System Architecture. The developer provides a set of polymorphic methods and provides a configuration file describing how these methods shall be called. At runtime, CAreDroid monitors the phone context and adapts the application behavior accordingly.

the definition of application-specific rules used to map methods to contexts (Section 3);

- We extend the Dalvik Virtual Machine in the Android OS to provide adaptation support for context-aware application. The resulting CAreDroid can transparently switch between polymorphic versions of application methods at runtime (Section 5);
- We provide two levels of complexity of the mapping between contexts to methods, (i) a binary criteria (called must fit) and a relaxed criteria (called best fit) (Section 5.2);
- We demonstrate how application developers can leverage CAreDroid to make applications context-aware with minimal disruptions to the standard application development process (Section 6).

The remainder of this paper is organized as follows. Section 2 introduces the system architecture of CAreDroid. Details of CAreDroid including its configuration, monitoring, and context adaptation algorithms are presented in Sections 3, 4, and 5 respectively. Section 6 shows the evaluation and case studies. Finally, we discuss some issues related to the design of CAreDroid and give conclusions in Sections 7 and 8, respectively.

2. SYSTEM ARCHITECTURE

The main objective of CAreDroid is to provide the application developer with support to easily build adaptation in context-aware applications. Hence, from a developer perspective, the design of CAreDroid needs to satisfy the following properties:

1. **Usability:** CAreDroid needs to add minimal overhead on the application developer at development time.
2. **Performance:** The adaptation engined needs to add minimal execution overhead when the application is running.

Motivated by these two design objectives, we designed CAreDroid as discussed in this section. A conceptual overview of the CAreDroid architecture is shown in Figure 1. Applications normally call polymorphic methods f and g. Each method is aliased to one of its versions (f_1 and g_2, respectively, in the example). A sensitivity configuration file, defined on a per-application basis, describes rules that determine under what context each of the polymorphic versions should be used. For each version of a method, sensitivity rules define acceptable ranges of operation for different sensors of system context. Method f_1 could define, for example, two rules stating that WiFi connectivity and battery charging status should be equal to true, while f_2 could define one rule stating that remaining battery capacity should be between 0% and 20%. Rules are assigned priorities that help determine which of the versions should be used when multiple rules are valid.

In the system layer, CAreDroid parses the application configuration file to discover adaptable methods and their rules of operation. A context-monitoring module abstracts the various sensors in the system, and exposes context information to an adaptation engine. When changes in context occur, the adaptation engine changes the aliasing of the adaptable methods according to the sensitivity rules. If more than one version of a method matches the current context, the priorities of the sensitivity rules are used to choose between them. When there are no alternatives of a method that exactly matches the context, CAreDroid chooses the version that most closely conforms to the current state of the device. CAreDroid is organized in three modules:

Context Sensitivity Configuration File

For each context-aware application, a sensitivity configuration file maps methods to contexts. The file is structured as a series of *sensitive methods* and their respective context *sensitivity lists* described in XML format. In keeping with our goal of decreasing development complexity for context-aware applications, the file is a straightforward description of acceptable ranges of operation for each method under different contexts. A detailed description of the CAreDroid configuration file is presented in Section 3.

Context Monitor

CAreDroid has a dedicated module that continuously probes the current phone context. CAreDroid supports both *raw contexts* that can be directly known by reading the state of the hardware (e.g. WiFi connectivity, battery level) as well as higher level *inferred contexts* such as mobility status (e.g., walking, running) that require advanced processing of sensor information. As mentioned in Section 1.1.2, the design of context monitoring systems is a well studied topic. The main work in this paper does *not* focus on efficient implementation of context monitoring system. However, context monitoring is yet an essential part in order to evaluate any adaptation engine. Hence, in Section 4 we describe a simplistic implementation of context monitoring which can be augmented by any of the previous proposed context monitoring algorithms.

Adaptation Engine

In order to choose the correct polymorphic implementation that best suits the current context, CAreDroid uses the data supplied by the developer in the configuration file. Alternative implementations of sensitive methods are connected together through a replacement map that lists all *candidates methods* that can be used for a sensitive call. Whenever more than one candidate implementation fits the current context, CAreDroid uses a conflict resolution mechanism to pick the implementation with the highest priority. Because it frees developers from having to implement adaptation strategies in the application layer, the CAreDroid adaptation engine is the main factor in meeting our goal of decreasing development complexity for context-aware appications. Section 5 shows how context-to-method matching and conflict detection are implemented efficiently to meet our goal of reducing runtime overhead.

3. SENSITIVITY CONFIGURATION FILE

The configuration file is an XML file that is supplied by the application developer. To fit in the Android flow, the configuration file is stored as an asset file packed with the application package file (APK). In this section, we describe the structure of this XML file along with the post-processing steps performed by CAreDroid over this file.

3.1 Configuration File Structure

For each *sensitive method*, the developer provides different polymorphic implementations. Each polymorphic implementation of a method is described by a name, a tag, and a priority. The name corresponds to the method name in source code. The tag associates different implementations of a method with one another. For example, if methods f_1 and f_2 are polymorphic implementations of the same method, then both of them must be associated with the same tag, for example f. Finally the priority for a method helps the system resolve ambiguities when multiple versions of a method satisfy the current context.

For each polymorphic implementation, the developer assigns a *sensitivity list*. This *sensitivity list* is the list of context states for which this polymorphic implementation shall be triggered. In our current implementation of CAreDroid, we focus on four context categories:

- **Battery state:** In this category, we define three contexts which are (1) the remaining battery capacity ($0\% - 100\%$) (2) the battery temperature (-30° C – 100° C) which is an indicative of high battery load as well as elevated power consumption; and (3) operating battery voltage, which is an indicative of the battery health.
- **Connectivity state:** In this category, we define three contexts: (1) WiFi connection status (On - Off), (2) WiFi link quality ($0-70$ A/V$^\circ$), and (3) RSSI Received signal strength indication ($0 - 4$).
- **Location:** In this category, we consider one context state, which is GPS location. In this state, the developer is allowed to provide the latitude and longitude coordinates of a square area.
- **Mobility state:** In this category, we consider only the current mobility state of the phone holder, which can take one of the following values: still, walking, running and driving.

```
<Method>
  <MethodName>AdjustCameraPowerAware
  </MethodName>
  <priority>1</priority>
  <tag>cameraAdjust</tag>
  <batterycapacity>
    <vstart>0</vstart>
    <vend>25</vend>
  </batterycapacity>
</Method>
<Method>
  <MethodName>AdjustCameraWhileRunning
  </MethodName>
  <priority>2</priority>
  <tag>cameraAdjust</tag>
  <batterycapacity>
    <vstart>20</vstart>
    <vend>100</vend>
  </batterycapacity>
   <mobility>run</mobility>
</Method>
```

Figure 2: Snippet of a CAreDroid configuration file.

3.1.1 Example

To illustrate the construction of the configuration file, we provide a small example in Figure 2. In this example, we have two polymorphic methods for adjusting the camera parameters under different contexts. One method, `AdjustCameraPowerAware`, is designed to save energy. Hence, its `BatteryCapacity` range is from 0% up to 25%, and it can execute whether `wifi` is on or off. The second method is dedicated to adjusting the camera while the user of the device is running. For example, this method should adjust the focus and the scene parameters of the camera to give a better quality image. Accordingly, the `mobility` is assigned to be *run*.

3.2 Configuration File Processing

After the developer supplies the CAreDroid configuration file, several post-processing steps are required at the installation time of the application. In particular, the XML file needs to be parsed, and the extracted information is then used to fill specific data structures. Figure 3 shows how CAreDroid flow extends the normal Android compilation and installation flow. This flow diagram shows the steps needed to post-process the configuration file. Parsing of the configuration file, discovery of sensitive methods, and registration of adaptation parameters with the adaptation engine is implemented in the Dalvik Virtual Machine (DVM), as described in the remainder of this section.

3.2.1 Sensitive Method Discovery

Upon compilation of the Java code, the generated Dalvik Executable File (DEX) contains all compiled bytecodes of methods stacked on top of each other. A call to a method is then accomplished by pointing to the offset of the first instruction inside the DEX file. For example, let us consider a call to the `myObject.foo()` method. The following bytecode:

```
invoke-virtual {v14}, [method@101e]
```

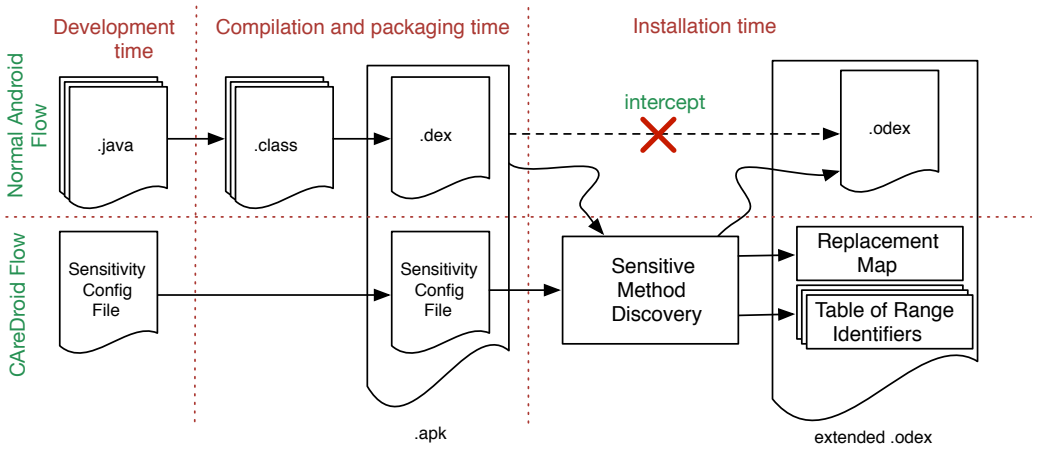


Figure 3: CAreDroid's extended installation flow. CAreDroid intercepts the installation process of the app on the device in order to parse the sensitivity configuration file. The final outcome of this process is two data structures named "Replacement Map" and "Table of Range Identifiers."

is used, where `v14` is the reference to the object instantiated from the class `myObject`, and `0x101e` is the offset of the first instruction in `myObject.foo()` in the DEX file. Note that the textual name of the method (e.g. "foo") is still preserved in the generated DEX file and the association between the method name and the method offset can still be extracted.

The DEX file along with all asset files (including the CAreDroid configuration file) are then packaged in the application package file (APK). When the APK file is installed, Android creates a new virtual machine to host the application. During this process, the Android flow extracts the DEX file and post-processes it in order to generate the Optimized DEX (ODEX) file.

The DEX optimization consists of two main steps. The first step is executed while class loading takes place. During this step, each method is assigned with a local method ID (compared to the global method ID assigned in the DEX). The second step of the DEX optimization process takes place when object references are linked with their classes. In this step, inheritance, polymorphism, method overriding, and method overloading are resolved. In particular, a virtual table is generated for each class. Each resolved method corresponds to an entry in this virtual table. Therefore, each method is now identified with its unique index inside it class virtual table. As a result, the call to the `myObject.foo()` method is further translated into:

```
invoke-virtual-quick {v14}, [000c]
```

where `v14` is the reference to the class object, `myObject`, and `000c` is the index for method `foo` inside that class's virtual table. Note that the association of the method name with its index in the virtual table is no longer preserved in the ODEX file.

Switching between different polymorphic implementations is equivalent to intercepting the operation of the bytecode corresponding to `invoke-virtual-quick` and supplying a different method ID. To perform this operation, CAreDroid must be able to keep track of the method IDs and relate them back to the IDs of different polymorphic implementations. Therefore, CAreDroid modifies the DEX/ODEX build process in Android to add hooks for context-awareness in sensitive method calls. This is accomplished through a *table of range identifiers* and a *replacement map*. This process is shown in Figure 3.

3.2.2 *Replacement Map (RM) and Table of Range Identifiers (TRI)*

The "Replacement Map" (RM) is a collection of (key, value) pairs defined for each polymorphic method.
The purpose of this map is to link each of the multiple polymorphic alternatives with their *sensitivity lists*. The *key* of this map is a composite key that consists of the pair of class-id and method-id extracted initially from the DEX file. The *value* field of the RM is an array whose length is equal to the number of contexts (mobility, location, battery capacity, etc.). This array specifies the *sensitive* operation range for this method for all different contexts. To facilitate this association, we introduce another data structure called the "Table of Range Identifiers" (TRI).

The TRI consists of multiple *associative arrays*. For each of the context sensors, we create a corresponding associative array. To construct such an array, we extract all the operation ranges provided in the configuration file, and and associate a uniquely generated *operation range identifier* (ORI) to each of the operation ranges. An example for such an associative array for battery capacity is shown in Table 1. Since the ranges of operations can vary from one class to another (based on the developer's intent , as described by the configuration file), we generate a TRI per class per context. The association between the class and the corresponding set of TRIs is made after the optimization of the DEX file process takes place. Once all TRI tables are built, we connect them to the RM by copying the corresponding ORI from the TRI data structure. An example of the RM is shown in Table 1.

At runtime, CAreDroid uses the TRI along with the current context to retrieve all ORIs that satisfy the current context. These ORIs are then used as inputs to the RM to retrieve the corresponding method IDs. If more than one method matches the ORIs, a conflict is discovered and needs to be resolved as described in Section 5.

B-Range	ORI
0→100	1
20→30	2
10→20	3
30→100	4

S-Range	ORI
0→2	1
1→4	2
2→3	3
0→3	4

...

Class ID	Method ID	B	T	V	W	Q	S	M	L
0x01	0x00F	1	2	1	2	4	2	1	4
0x01	0x01E	2	3	4	2	3	3	2	2
0x02	0x02A	2	2	1	0	0	0	2	1
0x02	0x01F	2	2	1	0	0	0	8	3

Table 1: On the left, examples of TRI tables for Battery capacity and Signal strength (RSSI) contexts. Each TRI associates a unique *Operation Range Identifier* (ORI) to each record. For each class, CAreDroid creates TRI for all different contexts. The association between the TRIs and the class is done later, after the optimization of the DEX files takes place. On the right, a Replacement Map that associates each key = (class-id, method-id) with its corresponding ORIs. CAreDroid creates a unique RM for the application. — *legend:* B: Battery Capacity, T: Battery Temperature, V: Battery Voltage, W: WiFi connectivity, S: Signal strength, Q: Signal Quality M: Mobility L: Location.

3.2.3 ODEX Extension

After CAreDroid constructs all the TRIs and the RM data structures, the DEX file passes through the normal Android optimization process, resulting in the generation of the ODEX file. Since the optimization process of the DEX file can result in a change in the method IDs, CAreDroid intercepts the process of optimizing the DEX files in order to update the RM, as shown in Table 1.

Finally, we extend the ODEX file structure by adding a reference to the RM data structure, which is generated by the described process. We extend the internal Android *class* data structure in order to associate the corresponding TRIs generated for that particular class. We also extended the internal Android *object* and *method* data structures by adding *sensitivity flags*. These flags are used later by the **CAreDroid Adaptation Engine** to facilitate the method switching.

3.3 Online Change of Context Ranges

While the configuration file needs to be specified by the developer at development time, the sensitive values of some sensitive contexts may not known until the code is running on the phone. For example, an application that changes its behavior whether the user is at *home* or at *work*. The location information (longitude and latitude of home and work) is not known at development time. Accordingly, CAreDroid supports online modification of the values associated with each sensitivity context. This takes place by asking the developer to write a specific file to the file system. CAreDroid parses this file whenever appropriate and re-updates the TOI accordingly. Note, that CAreDroid allows only changing the values associated with each sensitivity list but not the number of sensitive contexts associated to a polymorphic implementation.

4. CAREDROID CONTEXT MONITORING

In this section, we describe how CAreDroid acquires the current context at runtime with less overhead than Android Java APIs. Phone contexts can be numerous, and include raw values (like accelerometer data, GPS longitude and latitude, remaining battery capacity, etc.), or inferred states (like user mobility). While the in contribution of this paper is *not* efficient implementation of a context monitoring system, this is an essential part of any adaptation engine. In this section, we give details on how CAreDroid acquires both raw and inferred phone contexts. The work in this section can be indeed complemented by any of the context monitoring systems that appeared currently in the literature.

4.1 Raw Context Monitoring

Android exposes sensor information to the software stack through a Hardware Abstraction Layer (HAL). The HAL features a set of sensor managers that work as an intermediate layer between the low-level drivers and the high-level applications.

In order to reduce the overhead, we need to bypass the HAL layer and the associated sensor managers. This can be done by snooping on the interface between the HAL and the low-level device drivers through the `sysfs` virtual file system. In particular, each sensor (e.g. accelerometer, battery sensors, and WiFi) device driver exports its data into a set of files located under `/sys/class/`. In our work, we create an internal Dalvik VM thread that continuously reads these files to determine the state of the battery sensors and WiFi availability. The WiFi signal quality and signal strength are monitored via reading `/proc/net/wireless`. Similarly, the GPS location is determined by snooping over the Android Binder that connects the `Android LocationManager` with the GPS hardware driver.

4.2 Inferred Context: Mobility State

Mobility state detection is calculated by processing the raw accelerometer data obtained by the internal VM thread described above. In order to infer the mobility state, we adapt the classification procedure described in [28, 29] to detect whether the user is stationary, walking, or running. This classifier is based on the Geortzel algorithm [18]. Finally, to reduce the computational delay due to running the mobility state classifier, we let the classifier run on a separate DVM internal thread.

5. CAreDroid ADAPTATION ENGINE

The adaptation Engine is the core of CAreDroid. It is where the method replacement happens at runtime. In this section, we explain how CAreDroid extends the execution phase of the Android flow to automatically and transparently switch between methods.

5.1 Dalvik Interpreter Extension

Recall that the developed application is compiled and translated into an ODEX file. Bytecode stored in the ODEX file is then interpreted at runtime. In particular, the Dalvik Virtual Machine (DVM) runtime utilizes two types of interpreters. The first is called the *portable* interpreter, which is

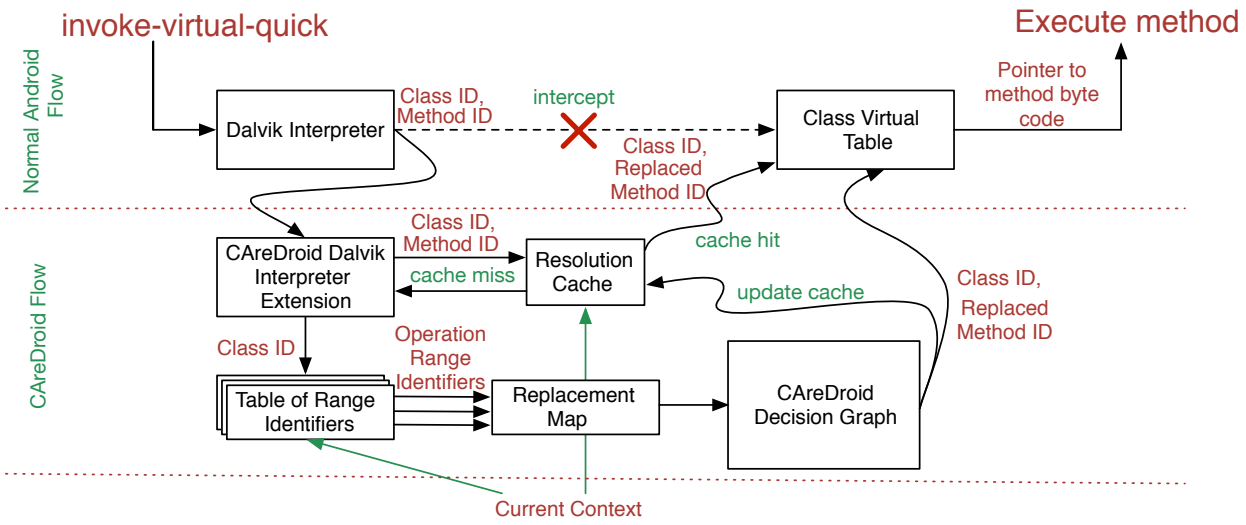


Figure 4: Flow of CAreDroid Adaptation Engine at runtime. The CAreDroid extended interpreter intercepts the execution of the Dalvik opcode *invoke-virtual-quick* to check whether the method invoked is sensitive or not. If the method is sensitive, then the CAreDroid adaptation engine checks the current context and picks the correct polymorphic method. This process is done through leveraging the information in the TRI and RM data structures along with the conflict resolution mechanism implemented using the decision graph. Finally, to speed up the process, CAreDroid uses a resolution cache, which exploits the temporal locality of the context.

implemented in C code and is not specific to a certain platform architecture. The second interpreter is called the *fast* interpreter which is implemented in assembly language and tailored towards a specific platform. The DVM supports switching between the two interpreters at runtime.

In our framework, we extend the *portable* interpreter to support the CAreDroid runtime engine. The extended interpreter checks the current interpreted ODEX bytecode. Whenever the bytecode corresponding to the `invoke-virtual` instruction is detected, CAreDroid intercepts the execution of the interpreter. It then checks the arguments of the `invoke-virtual` instruction — the method ID and the class ID — against the *sensitivity flags* in the extended ODEX file, described in Section 3.2.3. If the *sensitivity flag* is set, then CAreDroid needs to pick the polymorphic method that best fits the current context.

The process of choosing the best polymorphic implementation needs to resolve the conflicts in the user configuration. This is done through the CAreDroid decision graph module (discussed later) along with the TRI and RM data structures. In order to accelerate the process of picking the correct polymorphic implementation, CAreDroid uses a *resolution cache* that exploits the temporal locality of the adaptation decisions. This process is shown in Figure 4 and illustrated in the CAreDroid *decision graph* and the *resolution cache* in the following subsections.

Note that the *portable* interpreter (where CAreDroid is running)has a negative effect on the execution time of the application. To address this issue, we switch between the *fast* interpreter and the *portable* interpreter at runtime. The execution starts normally with the *fast* interpreter and, when the interpreter hits an invocation of a sensitive class, the interpreter switches to the *portable* interpreter and the CAreDroid adaptation process takes place. After executing the sensitive method, the interpreter switches back to the *fast* version.

5.2 Which Polymorphic Implementation to Pick?

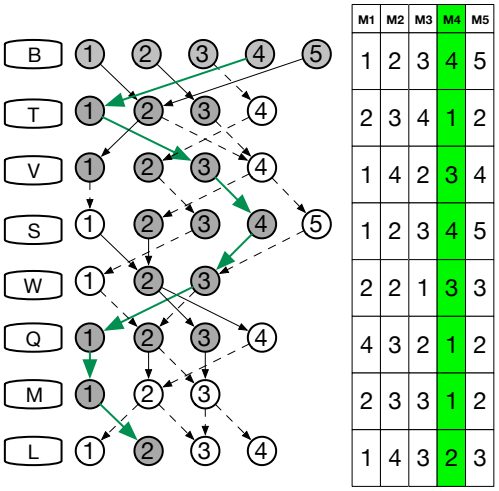
In order to choose the correct implementation that best suits the current context, our framework utilizes the data supplied by the developer in the configuration file. Note that it is possible that, for a given context, multiple methods are valid *candidates*, leading to a conflict that needs to be resolved.

5.2.1 *Best Fit vs Must Fit*

The first step is to choose a set of *candidate* methods. We allow for two policies. In the first policy, *must fit*, a method is considered a valid *candidate* if the current context satisfies *all* the operation ranges for all sensitive contexts. In the second policy, *best fit*, a method is a valid candidate if the current context satisfies *at least one* operation range of the sensitive contexts. The choice of policy is determined by the configuration file.

5.2.2 *Decision Graph*

We use a Directed Acyclic Graph (DAG) to choose the candidate method. Each level of the graph marks one sensitive context (e.g. battery capacity, mobility state). The sensitive contexts are ordered based on their priority as defined in the configuration file. For each sensitive context, we create nodes for all *operation range identifiers* (ORI)— previously discussed in Section 4— that appear in the *replacement map* (RM) data structure. In other words, to build the decision tree, we traverse the RM horizontally.



M1	M2	M3	M4	M5
1	2	3	4	5
2	3	4	1	2
1	4	2	3	4
1	2	3	4	5
2	2	1	3	3
4	3	2	1	2
2	3	3	1	2
1	4	3	2	3

Figure 5: An example of a Replacement Map(RM) (right) and its associated decision graph (left). The nodes at each level correspond to the ORIs in the same level of the associated RM. The edges in the decision graph correspond to the five methods shown in the RM. The shaded nodes correspond to the ORIs that match the current context. The solid arrows correspond to the active paths that match both the RM and the current context. Finally the path marked in green corresponds to the method that satisfies all the ORIs, and therefore this method is the one picked by CAreDroid for execution.

For each row of the RM, we create nodes corresponding to all distinctive ORIs in that row. This process is repeated for all the rows in the RM. An example of an RM and the associated decision graph is shown in Figure 5.

The methods contained in the RM columns dictate the decision graph topology. Accordingly, we traverse the RM vertically and connect the ORIs that correspond to the same method by edges. This is shown for the same example, in Figure 5.

When the phone context is reported, we use the TRIs in order to know which ORIs are active, i.e., which ORIs match the current context. The next step is to use this information to eliminate some choices in the decision graph. For example, in Figure 5, we mark the active ORIs with a gray color and the corresponding active edges with solid arrows.

The final step is to compare the available active paths that start from the top level. In the *must fit* policy, CAreDroid considers only the active paths that connect the first level all the way to the lower level. If no such path exists, then no method replacement is going to take place. On the other hand, the *best fit* policy considers the longest path that starts from first level. Referring to the example in Figure 5, only one active path is passing through all the DAG levels and corresponds to method M4. Therefore, CAreDroid picks this method for execution. The Class ID and Method ID of this method is reported back to the normal Android flow to be executed. If further conflict exits, we use the method priority reported in the configuration file.

Platform	SLOC	% Increase
Non-context aware (Base)	275	-
Context-aware (Pure Java)	606	120%
CAreDroid	275 +78[a]	28.3%

[a]SLOC of the XML Configuration file

Table 2: significant line of code (SLOC) results for case study 1 showing the SLOC for different implementations along with the percentage increase of SLOC relative to the non-context aware implementation.

5.3 Conflict Resolution Cache

While the adaptation strategy for CAreDroid is fairly straightforward, performing it for every individual sensitive method call in the system would incur in an unnecessary overhead. In order to decrease the overhead of the context-to-method resolution mechanism, our framework uses a resolution cache. Our heuristic assumes that the operating point does not change over short time periods. Therefore, if a method is called multiple times within a short amount of time (inside a loop for example), the same polymorphic implementation might be used for all of these calls. The cache is used to store the recently resolved candidates, that is, the recent phone context along the method ID that is chosen for each phone context. Each entry corresponds to the eight values of the phone context along with the method ID for the optimal method. The cache uses a Least Recently Used (LRU) approach to replace entries.

6. EVALUATION

We evaluate CAreDroid with four case studies. In the first one, we focus on assessing two metrics namely, reducing the number of significant lines of code and the execution time of the context-aware application. In the remaining three case studies, we show examples of applications that can benefit from context adaptation using CAreDroid.

All case studies are carried over a Nexus 4 phone running a modified system image for platform 4.2 API 17 [3]. The execution time is obtained using the Android SDK tracer [4]. The size of the original system image for Android 4.2 is 234.368 MB, the modified system image that support CAreDroid is 245.26 MB. Hence, the overhead in the system image is 4.6%.

6.1 Case Study 1: A Simple Application

In this case study, we implement a simple application that has only one sensitive method with three polymorphic implementations. In particular, this simple application implements a numerical solver for linear equations (which is a cornerstone algorithm in many image processing algorithms used to enhance photos before posting them to social media applications). We implement three polymorphic variants of this solver named LUP-decomposition (LU), Cholesky decomposition (CHD), and Conjugate Gradient (CG). These three methods have different memory and computation time characteristics. These mathematical functions are exhaustively used in image processing applications. Each implementation corresponds to a particular tradeoff between performance and computation time. In particular, CHD gives

Platform	Solver	CPU time (ms)						Overhead	
		Method time	Decision Tree		Context Monitoring (parallel thread)	Total			
			without cache	with cache		without cache	with cache	without cache	with cache
Non-context aware (Base)	LU	8.322	-		-	8.322		-	
	CHD	16.872	-		-	16.872		-	
	CG	13.375	-		-	13.375		-	
Context aware (Pure Java)	LU	8.322	0.227		5.093	13.642		63.92%	
	CHD	16.872	0.776		5.093	25.741		52.56%	
	CG	13.375	0.351		5.093	18.819		40.70%	
CAreDroid (Must Fit)	LU	8.322	0.183	0.030	0.336	8.841	8.688	6.23%	4.39%
	CHD	16.872	0.335	0.031	0.336	17.543	17.239	3.98%	2.17%
	CG	13.375	0.198	0.030	0.336	13.909	13.741	3.99%	2.736%
CAreDroid (Best Fit)	LU	8.322	0.183	0.031	0.336	8.841	8.689	6.23%	4.41%
	CHD	16.872	0.732	0.031	0.336	17.635	17.239	4.522%	2.17%
	CG	13.375	0.489	0.030	0.336	14.2	13.741	6.17%	2.73%

Table 3: Execution time results for case study 1 showing the profiling of different parts for all the three implementations. The overhead is computed relative to the non context-aware implementation. The results show the efficiency of both the must fit and best fit policies. It also shows the performance increase resulting from using the cache.

the most accurate results while suffers from high computation time. On the other extreme, LU gives the least accurate results (compared to CHD and CG) while leads to better computation time. The purpose of this case study is to characterize the performance of CAreDroid while switching between these three polymorphic implementations.

In order to characterize the CAreDroid performance, we generate an arbitrary configuration file that assigns each of the three solvers to different battery and connectivity contexts. We evaluate CAreDroid against a pure Java implementation performing the same functionality. That is, the pure Java application listens to changes in battery and WiFi connectivity using the standard HAL callback mechanism provided by the Android APIs. We implement a *non-context aware* implementation that *magically* knows which polymorphic method shall be called without knowing the context (for the purpose of comparison) and we call it the *base non-context aware*.

6.1.1 Reduction in Significant Line of Code (SLOC)

In this example, using CAreDroid reduces the SLOC for the application by a factor of 2x compared to a Java implementation using standard Android APIs. Table 2 shows the SLOC for each of the implementations.

6.1.2 Reduction in Execution Time

In this test case, we let the 4 different implementations (non context aware, pure Java, must fit and best fit) run over the phone for several hours while collecting profiling information. The profiling information are then averaged out and the result is reported in Table 3. To further investigate the effect of the resolution cache, we run the test with and without the cache functionality to allow for comparison. The results in Table 3 show that CAreDroid reduces the CPU time overhead (compared to the pure Java implementation) by a factor of 12x, on average, while adding a minimal overhead (2.5%–4.4%) compared to the non context-aware case. Furthermore, the resolution cache leads to decreasing the decision tree time by at least an order of magnitude whenever there is no change in the operating point. Finally, with no cache (or alternatively when a cache miss occurs) *best fit* policy adds slightly more overhead compared to the *must fit* policy due to the complexity of the decision graph used by the former. The same order of overhead also appears in the pure Java implementation because of the added code for switching between contexts.

6.1.3 Energy Profiling

Finally, we characterize the energy consumption (and hence the battery life time) due to context monitoring and adaptation. In this experiment, we monitor the voltage and discharging current of the battery during 2.5 hours while running the application under the four platforms (best fit, must fit, pure Java, and non-context aware). The experiment starts at the same battery capacity and at room temperature. We run each experiment three times. In the first one, we deactivate the context switching functionalities and focus only on the energy consumed by the context monitoring. These results are reported in Figure 6(a). In the second run, we run both context monitoring as well as context switching but calling an empty method. The energy measurements are then subtracted from the energy measurements from the previous experiment. The purpose of this experiment is to profile the effect of the decision tree and the context switching mechanism. This is shown in Figure 6(b). Finally, we run the full implementation to get the overall energy consumption of our system and compare it to the non-context aware one.

Overall, the results show that bypassing the HAL layer and performing the context monitoring inside the OS lead to decreasing the energy consumed by a factor of 36%. The results also show a similar decrease of energy consumption due to implementing the context switching inside the OS with a slight difference between the must fit and the best fit switching policies. Also, as seen in Figure 6(c), the energy consumption of pure Java implementation consumes around 69.33% more energy compared to CAreDroid. This energy consumption can be further improved by using energy-aware context monitoring techniques that previously reported in the related work (Section 1.1.2).

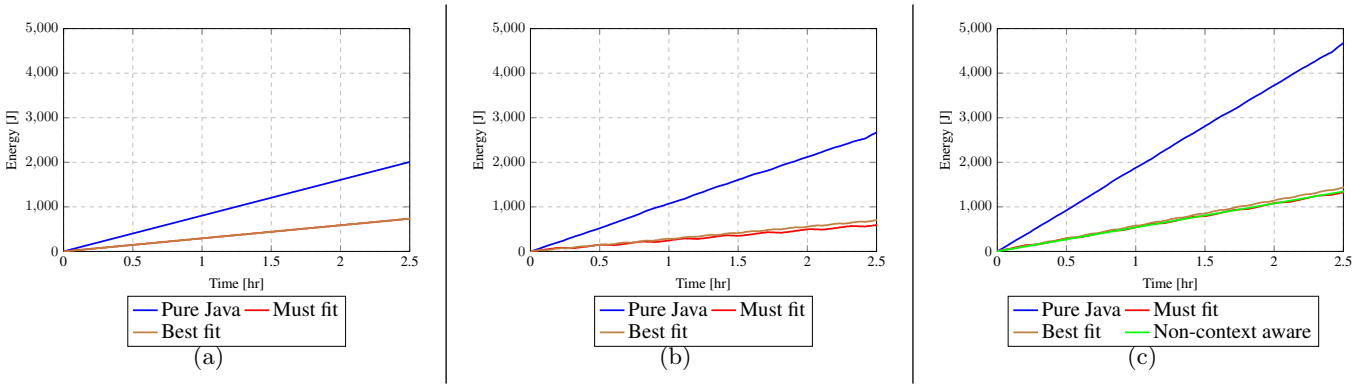


Figure 6: Energy consumption results for case study 1 showing (a) energy used when context monitoring is running alone (b) energy consumed by the context switching subsystem and (c) the total energy consumed. On each case, we plot the energy consumed by the pure Java implementation as well as the must fit and best fit implementations of CAreDroid. The results in (a) show that bypassing the HAL layer and implementing the context monitoring inside the OS allowed CAreDroid to use 36% less energy within the 2.5 hours lapse of the experiment. The results in (b) show that both Must fit and Best fit adaptation significant outperform the pure Java implementation in terms of energy consumed (and hence battery lifetime). The overall results (c) show that CAreDroid consumes only 6.73% energy compared with the non-context aware implementation and provides 69.33% energy saving compared to the pure Java implementation.

6.2 Case Study 2: A Context-Aware Phone Configuration

With increasing reported accidents resulting from texting while driving, we develop an application that changes the phone configuration based on the underlying context of the phone[1]. We manifest the **location**, **mobility state**, and **battery** in this application. In particular, we would like the application to change the phone configuration as follows:

- **Default**: keep the phone in its default configuration.
- **Driving**: (1) disable messaging and email notifications, (2) block certain caller numbers specified by a list (i.e. forward calls from this list to the voice mail) and (3) enable bluetooth (to connect the phone to car speaker).
- **Running**: (1) enable GPS (if not enabled), (2) block certain caller numbers specified by a list, and (3) mute the alarms.
- **At home**: (1) enable WiFi, (2) block certain caller numbers specified by a list, (3) raise the alarm volume, and (4) set ringer volume to normal.
- **At work**: (1) enable WiFi, (2) lower the alarm volume, (3) put the phone in vibrating mode, and (4) block certain list of caller numbers.
- **Power saving**: (1) lower the ringer volume, (2) disable bluetooth (if enabled), and (3) enable the automatic adjustment of screen brightness.

[1]Some applications in the market attempt to control the phone configuration like Tasker [2] and Locale [1] by providing hooks to the user to configure the phone based on certain rules that the user defines. However, these applications have only boolean decision. The rules must be all satisfied in order to change the configuration, while CAreDroid provides more complex formula (the best-fit policy). Moreover, Tasker and Locale do not support all the contexts supported by CAreDroid.

Platform	SLOC	% Increase
Non-context aware (Base)	282	-
Context-aware (Pure Java)	873	201%
CAreDroid	282 +277[a]	98.2%

[a]SLOC of the XML Configuration file

Table 4: Significant lines of code (SLOC) results for case study 2 showing the SLOC for the three implementations along with the percentage increase of SLOC relative to the non-context aware implementation.

Platform	CPU time[a]	Overhead
Non-context aware (Base)	1.942	-
Context aware (Pure Java)	12.14	525.12%
CAreDroid (Best Fit)	2.015	3.76%

[a]CPU Time (ms) = Method time + HAL Callback time + Inferences

Table 5: Execution time results for case study 2 showing the overhead for the different implementations.

For each of these configurations, a polymorphic method is implemented. The objective is to call the correct method based on the context. Similar to the previous case study, we implemented a non-context aware implementation (for the purpose of comparison), a context-aware implementation using the normal Android flow, and a context-aware implementation using CAreDroid.

6.2.1 *Reduction in Significant Line of Code (SLOC)*

CAreDroid decreases the code complexity (quantified by SLOC) by a factor of 2× (including the SLOC of the XML configuration file) compared to the implementation based on the normal Android flow, as shown in Table 4.

6.2.2 Reduction in Execution Time

In this test case, CAreDroid is configured and the phone is allowed to change between different contexts leading to a change in the application behavior. The time profiling is done across different contexts and the one with maximum CPU overhead is reported in Table 5.

For this case study, the pure Java implementation adds a 525.12% CPU overhead. On the other hand, CAreDroid introduces a minimal overhead of 3.76% compared to the non-context aware implementation. The large overhead of the former can be explained by observing that there are 16 possible cases that need to be handled if the application developer were to implement the same app without using CAreDroid. Needless to say that the developer—without CAreDroid—has to implement all the Android listeners to all contexts as well as provide the high-level inferences of mobility state from the raw data. The small overhead in CAreDroid compared to the pure Java implementations again can be accounted to the fact that all the context-awareness operations (context monitoring and adaptation) are implemented natively inside the operating system.

6.3 Case Study 3: Context-Aware Camera

In this case study, we build a context-aware camera application. The camera adjusts its features parameters based on the phone context. We have five different methods that CAreDroid alternates between. The focus of this study is on making the *focus*, *scene mode* and *flash mode* adaptive to the context. However, this can be extended to handle all the camera features. The five modes are listed as follows:

- **Default**: Configure the focus mode to "default"
- **When runing**: adjust the focus mode to the "continuous picture" mode.
- **When walking**: adjust the scene mode to the "steady photo" mode.
- **When still**: adjust the focus mode to the "fixed" mode".
- **Power saver**: (1) adjust the flash mode to "off", (2) adjust the focus mode to "fixed" mode, and (3) adjust the quality of the picture to "minimum."

Similar to previous case studies, we implemented a polymorphic method for each of these modes and the objective is to call the appropriate method based on the phone context.

Platform	SLOC	% Increase
Non-context aware (Base)	277	-
Context-aware (Pure Java)	782	182%
CAreDroid	277 +133[a]	48%

[a]XML Configuration file

Table 6: Results of the significant line of code (SLOC) for the three implementations of the Smart Camera application used in case study 3. The results shows the SLOC along with the percentage increase relative to the non-context aware implementation.

The test is performed as follows. First a photo is captured while the phone is held in a standstill position using the original *camera* application provided by the phone. Next, the user starts to walk/run while trying to capture the photo for the same object again using the original *camera* application. Finally, the same experiment is done while using the CAreDroid-based *context-aware camera* application.

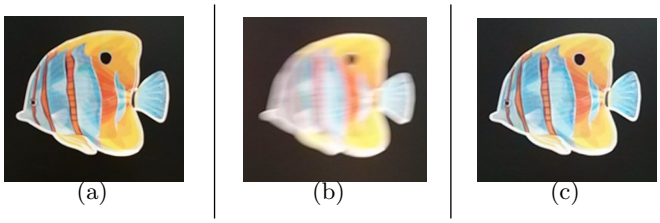

(a) (b) (c)

Figure 7: Photos taken by the Smart Camera application developed for case study 3: (a) the photo taken while the phone holder is standing still, (b) the photo taken while the phone holder is walking and no context-awareness is taking place, and (c) photo taken while the phone holder is walking and using the Smart Camera application built on top of CAreDroid.

The results of the implemented application is shown in Figure 7. Figure 7(a) shows the original photo captured from a stand still position. Figure 7(b) shows the captured photo while the phone holder is walking and no context-awareness processing is taking place, and finally, Figure 7(c) shows the captured photo with the user is walking and using the developed context-aware camera with CAreDroid. As with the previous case studies, Table 6 shows the reduction of the overhead in SLOC when the context-aware Camera application is developed using normal Android flow compared to the proposed CAreDroid flow. The table shows a reduction of SLOC by more than a factor of 3×.

7. DISCUSSION

The underlying idea behind CAreDroid is the ability of the system to sense and adapt to variations in the environment and available resources. In this section we discuss some issues that faced us during the design and implementation of CAreDroid.

7.1 Why is CAreDroid implemented inside the OS?

One possible design of CAreDroid was to design it as a library which provided context adaptation functionalities through a set of exposed APIs. Compared to the current design of CAreDroid, the library-based design falls behind in terms of the two design criteria discussed in Section 2 named *Usability* design and *Performance*. From the usability point of view, the library implementation of the adaptation engine forces the developer to issue subsequent calls to the library APIs. Missing calls to the library APIs may result to degradation in the context awareness of the developed application. On the other hand, the current design of CAreDroid makes the application developer completely *oblivious* from the adaptation. He is asked only to provide the adaptation *policy* in the XML configuration file. Afterwards, CAreDroid intercepts the execution of the methods while it is being interpreted by the Dalvik VM and perform the adaptation automatically. From the performance point of view, implementing the context adaptation and monitor-

ing in the low level results into less execution time as proved by the experimental test cases shown in Section 6.

7.2 Privacy

Sensing and understanding the user's context and taking decisions accordingly can lead to various privacy leaks. Android privacy mechanism depends on providing the user with different queries in order to grant permissions to the application to use the sensory data. In our work, CAreDroid ensures that the adaptation policy specified by the developer does not use sensory data that are not permitted by the user. For this end, CAreDroid parses the application's permissions included in the Android *manifest file*[2]. The adaptation engine in CAreDroid uses only permissible contexts as per application permissions.

7.3 Developer Matters

Despite the fact that the adaptation engine decision is obfuscated from the developing phase, in some scenarios—for example when the best fit policy is used—the developer may be interested in retrieving the current operating point. Therefore, CAreDroid addresses this issue by providing an API called "*read_operating_point*()" which can be used to read the current values of different contexts.

7.4 Limitations

The CAreDroid framework described here is not without some limitations:

- Polymorphic methods in CAreDroid must be pure functions, i.e., they cannot perform I/O and cannot change global program states, and their output must depend only on the method arguments. To allow for non-pure functions, the framework would require state-migration procedures between every possible pair of polymorphic methods.
- CAreDroid assumes that an application developer can provide multiple implementations of sensitive methods. Specifying the right constraints is not an easy task and it may be better to suggest the right constraints to the developer during a validation phase of the application. However, this is an open research point and previous work [14, 8] has identified the importance of enforcing the developer to suggest the adaptation policy and not letting the adaptation engine automatically synthesize the adaptation policy.
- CAreDroid also expects the application programmer to be aware of suitable ranges of operations for different sensitive methods. In the future, we intend to explore automated code profilers that could suggest ranges of operation for each of the choices, helping users in defining suitable adaptation configuration files.

7.5 Broader Uses of CAreDroid

CAreDroid supports connectivity context such as Wifi connectivity, signal strength and quality of the signal as well as low level context such as battery temperature. These contexts can be manifested to decide if some intensive computation should be *offloaded* to a server or if an *approximate* computation should be used. In particular, if battery capacity is good (high enough) and there is WiFi connectivity with good strength then CAreDroid can switch to a method that offloads intensive computation to a server and remove the burden of computation from the phone. Hence, the concept of cyber-foraging discussed in Section 1.1.1 can be directly implemented using CAreDroid. Similarly, approximate computation (or algorithmic choice as discussed in Section 1.1.1) can be implemented using CAreDroid by manifesting the temperature context as well as the battery capacity context.

[2]an Android XML file that declares the permissions required by the application

8. CONCLUSION

Context-aware computing is a powerful technique for physically coupled software. It can enhance functionality and improve resource usage of applications by adapting them to context. In this paper, we present CAreDroid, an adaptation framework for context-aware applications in Android. CAreDroid allows applications developers to develop context-aware applications without having to deal directly with context monitoring and context adaptation in the application code. In CAreDroid, multiple versions of methods that are sensitive to context are dynamically and transparently replaced with each other according to application-specific configuration file. By pushing the context monitoring and adaptation functionalities to the Android runtime, CAreDroid is able to provide context-awareness more efficiently and with significantly fewer lines of code compared to current Android development flow. In particular, using different case studies, we show how CAreDroid can be used to develop context-aware applications. Results from these case studies show that CAreDroid reduces the code complexity by at least half while decreasing the computation overhead by at least a factor of 10× compared to non-CAreDroid applications.

Acknolwedgment

This research is funded in part by the National Science Foundation under grant CCF-1029030, by the Center for Excellence for Mobile Sensor Data-to-Knowledge under National Institutes of Health grant #1U54EB020404, and by the U.S. ARL, U.K. Ministry of defense (MoD) under Agreement Number W911NF-06-3-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

9. REFERENCES

[1] Locale for android. http://www.twofortyfouram.com.
[2] Tasker for android. total automation for android. http://tasker.dinglisch.net/index.html.
[3] Android. Android open source project. https://source.android.com.
[4] Android SDK. Profiling with traceview. http://developer.android.com/tools/debugging/.
[5] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: a language and compiler for algorithmic choice. *SIGPLAN Notices*, 44:38–49, June 2009.

[6] K. Ariyapala, M. Conti, and C. Keppitiyagama. Contextos: A context aware operating system for mobile devices. In *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, pages 976–984, Aug 2013.

[7] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. *SIGPLAN Notices*, 45:198–209, June 2010.

[8] G. Bai, L. Gu, T. Feng, Y. Guo, and X. Chen. Context-aware usage control for android. In *Security and Privacy in Communication Networks*, pages 326–343. Springer, 2010.

[9] R. K. Balan, M. Satyanarayanan, S. Y. Park, and T. Okoshi. Tactics-based remote execution for mobile computing. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 273–286. ACM, 2003.

[10] A. Beach, M. Gartrell, X. Xing, R. Han, Q. Lv, S. Mishra, and K. Seada. Fusing mobile, sensor, and social data to fully enable context-aware computing. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*, pages 60–65. ACM, 2010.

[11] L. Capra, G. S. Blair, C. Mascolo, W. Emmerich, and P. Grace. Exploiting reflection in mobile computing middleware. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):34–44, Oct. 2002.

[12] D. Chu, A. Kansal, J. Liu, and F. Zhao. Mobile apps: It's time to move up to condos. In *13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*. USENIX, May 2011.

[13] D. Chu, N. D. Lane, T. T.-T. Lai, C. Pang, X. Meng, Q. Guo, F. Li, and F. Zhao. Balancing energy, latency and accuracy for mobile sensor data classification. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys '11, pages 54–67, New York, NY, USA, 2011. ACM.

[14] M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich. Crêpe: A system for enforcing fine-grained context-related policies on android. *Information Forensics and Security, IEEE Transactions on*, 7(5):1426–1438, 2012.

[15] E. De Lara, D. S. Wallach, and W. Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *USENIX Symposium on Internet Technologies and Systems - USITS*, volume 1, pages 14–14, 2001.

[16] J. Flinn, S. Park, and M. Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 217–226. IEEE, 2002.

[17] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[18] G. Goertzel. An algorithm for the evaluation of finite trigonometric series. *American mathematical monthly*, pages 34–35, 1958.

[19] S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song. Seemon: Scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, pages 267–280, New York, NY, USA, 2008. ACM.

[20] A. Kansal, S. Saponas, A. B. Brush, K. S. McKinley, T. Mytkowicz, and R. Ziola. The latency, accuracy, and battery (lab) abstraction: Programmer productivity and energy efficiency for continuous mobile context sensing. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 661–676, New York, NY, USA, 2013. ACM.

[21] A. Lachenmann, P. J. Marrón, D. Minder, and K. Rothermel. Meeting lifetime goals with energy levels. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pages 131–144, New York, NY, USA, 2007. ACM.

[22] X. Li, M. Garzaran, and D. Padua. Optimizing sorting with machine learning algorithms. In *Proceedings of Parallel and Distributed Processing Symposium*, March 2007.

[23] T.-Y. Lin, T.-A. Lin, C.-H. Hsu, and C.-T. King. Context-aware decision engine for mobile cloud offloading. In *Wireless Communications and Networking Conference Workshops (WCNCW), 2013 IEEE*, pages 111–116, April 2013.

[24] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell. The jigsaw continuous sensing engine for mobile phone applications. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 71–84, New York, NY, USA, 2010. ACM.

[25] S. Nath. Ace: Exploiting correlation for energy-efficient and continuous context sensing. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[26] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. *SIGOPS Oper. Syst. Rev.*, 31(5):276–287, Oct. 1997.

[27] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos. Context aware computing for the internet of things: A survey. *Communications Surveys Tutorials, IEEE*, 16(1):414–454, First 2014.

[28] S. Reddy, J. Burke, D. Estrin, M. Hansen, and M. Srivastava. Determining transportation mode on mobile phones. In *Wearable Computers, 2008. ISWC 2008. 12th IEEE International Symposium on*, pages 25–28. IEEE, 2008.

[29] J. Ryder, B. Longstaff, S. Reddy, and D. Estrin. Ambulation: A tool for monitoring mobility patterns over time using mobile phones. In *Computational Science and Engineering, 2009. CSE'09. International Conference on*, volume 4, pages 927–931. IEEE, 2009.

[30] M. Satyanarayanan. Pervasive computing: Vision and challenges. *Personal Communications, IEEE*, 8(4):10–17, 2001.

[31] B. Schilit and M. Theimer. Disseminating active map information to mobile hosts. *Network, IEEE*, 8(5):22–32, Sept 1994.

[32] B. C. Smith. *Procedural reflection in programming languages*. PhD thesis, Massachusetts Institute of Technology, 1982.

[33] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: a language and runtime system for perpetual systems. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pages 161–174, New York, NY, USA, 2007. ACM.

[34] N. Vallina-Rodriguez and J. Crowcroft. Erdos: Achieving energy savings in mobile os. In *Proceedings of the Sixth International Workshop on MobiArch*, MobiArch '11, pages 37–42, New York, NY, USA, 2011. ACM.

[35] X. Zhao, Y. Guo, Q. Feng, and X. Chen. A system context-aware approach for battery lifetime prediction in smart phones. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 641–646, New York, NY, USA, 2011. ACM.