UNIVERSITY OF CALIFORNIA

Los Angeles

Context-Aware Runtime Engine For Android Operating System

A thesis submitted in partial satisfaction

of the requirements for the degree

Master of Science in Electrical Engineering

by

SALMA HOSNI EMAM MOHAMED ELMALAKI

2014

Abstract of the Thesis

Context-Aware Runtime Engine For Android Operating System

by

SALMA HOSNI EMAM MOHAMED ELMALAKI

Master of Science in Electrical Engineering

University of California, Los Angeles, 2014

Professor Mani B. Srivastava, Chair

Context-awareness—the ability of systems to sense and react to their environment—can enable software to intelligently adapt its functionality across time, locations, and operating conditions. In this thesis, we introduce CAreDroid, a Context-Aware runtime adaptation system for Android. CAreDroid allows the runtime system to switch between blocks of code that perform alternative or comparable functionality. At any point in time, CAreDroid activates the blocks of code that best fit the device's context. CAreDroid targets both the Java and native layers of Android and can help users develop and analyze tradeoffs between accuracy and cost in different contexts. The proposed framework enables applications to dynamically and transparently adapt their functionality, quality, and timeliness of results, leading to a range of 3–12x reduction in execution time compared to context-aware Android apps that do not rely on CAreDroid.

Context-aware operation is particularly important for energy efficiency. In mobile phones the marginal energy cost of performing any operation changes according to device context as determined by, for example, remaining battery capacity, connectivity status, or sources of environmental and process variability like operating temperature. We show that context-aware applications using CAreDroid can achieve 20%–40% energy savings with acceptable quality degradation.

The thesis of SALMA HOSNI EMAM MOHAMED ELMALAKI is approved.

William J. Kaiser

Puneet Gupta

Mani B. Srivastava, Committee Chair

University of California, Los Angeles

2014

To Prophet Muhammad, My Parents, Husband, Son, and My Elder Brothers.

Table of Contents

List of Figures

List of Tables

ACKNOWLEDGMENTS

All praise be to Allah the High, “who teacheth by the pen, teacheth man that which he knew not.”, Quran[96:4, 96:5].

I say what Prophet Solomon said: “··· O my Lord! so order me that I may be grateful for Thy favours, which thou hast bestowed on me and on my parents, and that I may work the righteousness that will please Thee: And admit me, by Thy Grace, to the ranks of Thy righteous Servants.”, Quran[27:19].

I would like to personally thank Lucas Wanner for the tremendous help and guidance he has given me in the completion of this Masters thesis. I would also like to thank all of my mentors that had a role in shaping my education and the Networked and Embedded Systems Lab for giving me years of invaluable experience.

I would like to thank my parents for their unfailing help along my age and for their efforts during this thesis development. They offered me much advice and encouragement that was a great source of comfort.

Finally, To my beloved husband: Thank you for your patience, great support and encouragement during the most important stages of my life. Thank you for taking all responsibility of our life providing me with full comfort and concentration in my work. I now have to repay you the countless nights and weekends spent in the working on this thesis.

CHAPTER 1

Introduction

1.1 Context-Aware Computing

Unlike in traditional desktop and server systems, in mobile and wearable devices the underlying execution context—defined for example by connectivity availability, remaining battery capacity, charging status, or user location—is ever changing. Context-aware computing [ST94] argues that, as this context changes, software functionality should change accordingly.

At the heart of context-aware computing is the ability to sense the environment and adapt accordingly. For each particular application, a set of contexts that affects performance is discovered, specialized context-sensing mechanisms are designed, and adaptation procedures are deployed. These context-aware computing techniques have been shown to improve the performance and efficiency of applications significantly [BGX10, ZGF11, PZC14, LLH13].

Context-awareness is important not only from a purely functional standpoint, but also has crucial implications on related aspects of computing. In particular, it is important to study how *context-aware* computing can be used to enhance *energy-aware* computing [RSH08, RZ11]. The problem of energy efficiency on mobile devices has often been attacked from the angle of trying to produce high quality and accurate results while spending less energy. We argue that, when a desire for powerful processors, advanced applications, and device longevity go hand in hand, compromises in quality must be made. Energy management actions should likewise be informed by device state and context.

1.2 CAreDroid

While previous work focused on application-specific context-aware implementations, systematic support for context awareness is still missing. In this paper we focus on runtime support for enabling context aware computation. In particular, we introduce CAreDroid, a Context-Aware runtime adaptation framework for the Android mobile operating system.

CAreDroid is loosely related to polymorphic engines [Yet93] in that it is used to transform sections of a program into different versions with alternate code paths that provide roughly the same functionality. Polymorphic engines are used to intercept and modify code transparently, typically for malicious purposes such as hiding malware functionality from anti-virus software. In CAreDroid, the different code paths provide functionality that is better suited to the device under its various operating contexts. Mutations from one version to another are triggered by changes in device operation context. CAreDroid can thus be seen as a dynamic or flexible version of delegation [Lie86] for object-oriented systems, where lookup rules are used to bind methods to objects. A specific block of code may be activated, for example, when processor temperature reaches a certain threshold (an indicator of higher leakage power consumption). A second block may be activated when remaining battery capacity drops under a specified percentage. In this work, the different code blocks may be either standard library functions provided by the runtime system or alternative implementations provided by application programmers. Per-application configuration files determine when and under what circumstances code mutations should be triggered. The runtime system monitors device context and transparently triggers mutations at appropriate times.

CAreDroid targets both the Java application layer and the Linux runtime libraries in the Android system. For simplicity, in this work we refer to these as the Java and native layers. In both layers, individual functions in applications and shared libraries can be marked as *sensitive* to device context. Polymorphic implementations of these functions are tailored to specific operating points for the device and activated according to a sensitivity list. For each of the polymorphic implementations, a sensitivity list determines ideal ranges of operation under different context vectors of energy expenditure (remaining battery capacity, temperature, and battery voltage) and connectivity (WiFi connectivity state, and signal strength, and signal quality). Calls to sensitive methods are identical to calls to regular functions, and the CAreDroid runtime monitoring system dynamically and transparently triggers adaptations to find the set of functions that, at any point in time, better suit the device's context.

1.3 Related Work

Context-awareness relies on the ability of a software system to switch between different code paths based on the context. The idea of alternate code paths, or *algorithmic choice* has been explored in the energy-aware software literature. Petabricks [ACW09] and Eon [SKG07], for example, feature language extensions that allow programmers to provide alternate code paths. The runtime system dynamically chooses paths based on energy availability. In Petabricks, multiple versions of object code are created and profiled for execution time and quality using a sample set of input data. Paths for an application may be chosen statically or altered in runtime through accuracy valuations. A similar process is used in Green [BC10], where a combination of a calibration phase and runtime accuracy sampling are used by the application to define which function to execute from a set of possible candidates. In Eon, the runtime system dynamically chooses paths based on energy availability. Levels is an energy-aware programming abstraction for embedded sensors based on alternative tasks [LMM07]. Programmers define task levels, which provide identical functionality with different quality of service and energy usage characteristics. The run-time system chooses the highest task levels that meet the required battery lifetime.

Algorithms and libraries with multiple implementations can be matched to an underlying hardware configuration to deliver the best performance [FJ05, LGP07]. Such libraries can be leveraged to choose the algorithm that best tolerates the predicted context variation and deliver a performance satisfying the quality-of-service requirement. In CAreDroid we leverage the notion of algorithmic choice from previous work and extend it to encompass a broader view of device context. While in previous work adaptation has been either left to applications or triggered by energy availability or CPU utilization, in CAreDroid adaptation can be triggered by multiple dimensions of a device context's state including both connectivity vectors (e.g., signal strength or WiFi availability) as well as energy expenditure vectors (e.g, temperature or remaining battery capacity). Adaptation in CAreDroid is transparent but subject to per-application mutation rules dictated by sensitivity lists for polymorphic methods.

1.4 Thesis Contribution

We show how CAreDroid can help users in developing and analyzing tradeoffs between accuracy and cost in different contexts, as well as reduce CPU utilization and energy consumption with user-defined quality degradation policies. Case studies using the Java and native layers show how applications can use CAreDroid to adapt to system context without having to directly deal with hardware sensors or manage multiple software implementations. Microbenchmarks show that the system can monitor hardware and environmental context and trigger code mutations with negligible overhead. We make the following contributions:

- We extend the Dalvik Java Virtual Machine in the Android layer and provide shared libraries in the Linux layer to support context-aware application execution that can transparently switch between polymorphic versions of functions at runtime;

- We design a framework for the implementation of context-aware polymorphic functions and for the definition of application-specific function mutation rules;
- We implement a context monitoring and mutation engine that finds the set of functions that best match a device's context at any point in time;
- We demonstrate how application developers can leverage CAreDroid to make applications context aware with minimal disruptions to the standard application development process.

1.5 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 presents the CAreDroid system architecture, including its configuration, monitoring, and context adaptation strategies. Chapters 3, 4, and 5 present implementation details CAreDroid. Chapter 6 shows evaluation and case studies. Finally, we present some discussion on Chapter 7 and conclude in Chapter 8.

CHAPTER 2

System Architecture

2.1 Overall Architecture

We propose CAreDroid as an application runtime support system where the operating system adjusts functionality according to context-aware policies. Applications and system libraries define multiple implementations alternatives $f_1, f_2, \cdots, f_n$ for a function f. A series of sensitivity rules dictates which of the alternatives of f should be used under different execution contexts. For example, f_1 could be used when the device is charging and has WiFi connectivity, and f_2 could be used when remaining battery capacity is below a 20%. The CAreDroid adaptation runtime monitors various sensors of device context and dynamically aliases f to the alternative that best fits device context at any time.

Figure 2.1 shows a conceptual overview of the CAreDroid architecture. Applications call polymorphic functions f and g normally. Each function 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. Function f_1 could define, for example, two rules rule stating that WiFi connectivity and charging status should be equal to 1, 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.

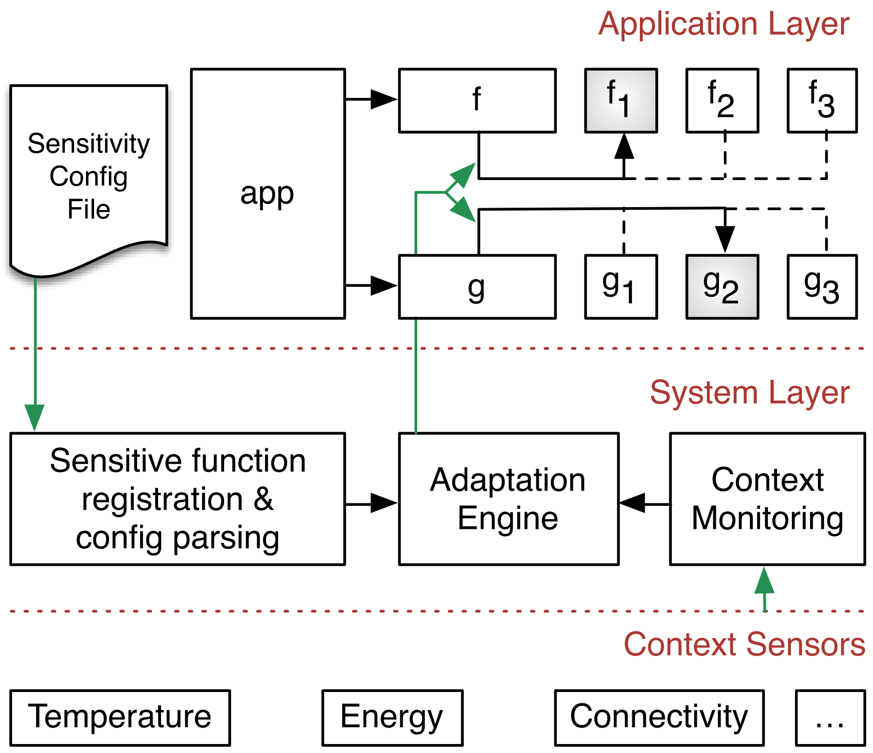


Figure 2.1: CAreDroid System Architecture

2.2 Sensitivity Configuration File

The sensitivity configuration file constrains and directs context adaptation for individual applications in CAreDroid. The file is structured as a series of *sensitive methods* and their respective context *sensitivity lists* described in XML format.

Each polymorphic implementation of a method is described by name, a tag, and a priority. The name corresponds to the function name in source code. The tag associates different implementations of a method to each other. For example, methods f_1 and f_2 could be associated with a tag f. For the Android/Java layer of CAreDroid, sensitive methods are encompassed by a sensitive class definition. Finally the priority for a method helps the system resolve ambiguities when multiple version of a method satisfy the current context. More information about the structure of the configuration file is given in Chapter 3.

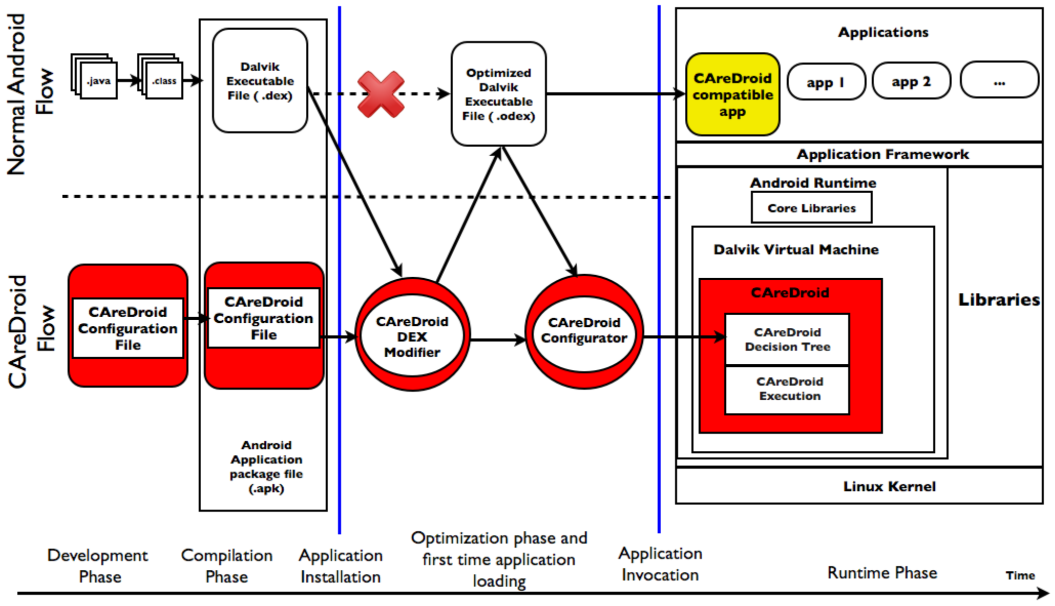


Figure 2.2: CAreDroid Tool Flow

2.3 System Layer

In the system layer, CAreDroid parses application configuration files to discover adaptable functions and their rules of operation. A context monitoring block abstracts the various sensors in the system, and exposes context information to an adaptation engine. When significant changes in context occur, the adaptation engine changes the aliasing of the adaptable functions according to the sensitivity rules. If more than one version of a function matches the current context, the priorities of the sensitivity rules are used to chose between them. When there are no alternatives of a function that exactly match the context, CAreDroid choses the version that most closely conforms to the current state of the device. Detailed information about the System Layer is given in Chapters 4 and 5.

Sensor	Description	Range	Unit
Battery	Remaining capacity	0–100	%
Temp.	Battery operating temperature	-30–100	°C
Voltage	Battery operating voltage	3–4.2	V
WiFi	Connection Status	0,1	–
Signal	Link Quality	0–70	A/V[a]
RSSI	Signal strength	-100 – -55	db

[a]Absolute value dependent on hardware/driver implementation. It is based on the contention, interference and frame error rate of the link.

Table 2.1: Context Sensors and Ranges.

2.4 Context Sensors

Ranges of operation for the various context sensors are defined for each method with *sensitivity lists*. Sensitivity lists define permissible ranges of operation for the vectors of sensitivity for each version of a method. While CAreDroid can support any available context sensors, for our prototype implementations we target a set of six energy and connectivity sensors shown in Table 2.1. The battery sensor indicates remaining battery capacity. The battery temperature sensor is indicative of high battery load as well as elevated power consumption. Operating voltage is indicative of battery health. The WiFi and link quality sensors indicate respectively whether a WiFi connection is available and its quality. Finally, the Received signal strength indication (RSSI) indicates the strength of the signal. This RSSI value is then used to set five levels for the signal strength from 0 (the lowest signal strength) to 4 (the highest signal strength).

CHAPTER 3

CAreDroid Configuration File

The CAreDroid Configuration File is the entry point for the proposed framework. It represents the interface between the developer and the CAreDroid framework. The developer is responsible of providing the necessary information on the context-based polymorphic methods. This information is manifested later in the lower layers.

3.1 Configuration File Structure

Listing 3.1: The schema of the CAreDroid XML configuration file.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Class>
   <ClassName> </ClassName>
   <Preference> </Preference>
   <Method>
      <MethodName> </MethodName>
      <orderOfPreference>  </orderOfPreference>
      <tag>  </tag>
      <item>
         <itemName>  </itemName>
         <vstart>  </vstart>
         <vend>   </vend>
      </item>
</Class>
```

The CAreDroid configuration file is an XML file. This file specifies the names of:

1. **Sensitive Classes:** the Java classes that contains context-based polymorphic methods.

2. **Sensitive Methods:** for each *sensitive class*, this field lists the method names that are prune to polymorphic, context-based replacements.

3. **Sensitivity List:** for each *sensitive method*, this field describes the different contexts for which this sensitive method needs to change its behavior. In particular, our current framework supports six contexts. Three of them are power-related (Battery capacity, battery temperature, and battery voltage) while the other three contexts are connectivity-based (Wifi availability, Wifi signal strength, and GPS availability).

The structure of the configuration file is shown in Listing 3.1. The configuration file contains multiple root elements, **Class**, that provide the required description needed by the CAreDroid framework. The **Class** element contains three different child elements: **ClassName**, **Preference**, and **Method**. In the remainder of this section, we are going to illustrate each of these child elements.

The **ClassName** element captures the name of a *sensitive class* in the application. Each application is allowed to contain several *sensitive classes*. All *sensitive classes* needs to be defined in the same configuration file.

The CAreDroid framework handles two different modes named, battery-aware execution and connectivity-aware execution. The **Preference** element of the XML file is used to switch between these two execution modes. In the battery-aware execution, the CAreDroid framework gives higher preference to the battery related context signals, like battery voltage and battery capacity, more than the connectivity related context signals like WiFi and GPS. On the other hand, the connectivity-aware execution, gives higher attention to the connectivity related context signals compared to the battery related signals. This preference is used as in input to the CAreDroid Runtime layer later. More information about these two modes of execution are presented in Chapter 5.

The third element in the configuration file is the **Method** element. In contrast to the previous XML elements, the **Method** element can appear multiple times, depending on the number of *sensitive methods* within the class. Each **Method** element corresponds to one implementation that can be polymorphically switched, on run-time, with other **Method** elements. Although, this definition may lead restricts each class to have only one set of context-based, polymorphic methods, we resolve this restriction using the further XML elements in the configuration file, as discussed below.

Each **Method** element contains a set of sub-elements named, **MethodName**, **orderOfPreference**, **tag** and **item**. The describe the *sensitivity list* associated with this *sensitive method*.

The **orderOfPreference** gives the *sensitive method* a rank. This rank is used to resolve ambiguity and contradictions that can appear in the configuration file. For example, a configuration file may contain two sensitive methods with overlapping values for their *sensitivity list*. In this case the lower the rank, the highest the priority vote for this method to execute under conflict.

The structure of the configuration file that we described so far specifies the names of the *sensitive classes* and *sensitive methods*. However, it is essential to categorize the *sensitive methods* into disjoint groups. A *sensitive method* can be switched with only the methods that belongs to its same group. That is why we introduced the **tag** element in the XML configuration file. The **tag** is used to mark the methods that are associated with each other as alternatives by assigning the same group number to all of them.

Finally, the **item** element is the one that specifies the sensitivity list for each *sensitive method*. Our framework handles three energy related contexts (battery capacity, battery temperature, and battery voltage) and three connectivity related contexts (WiFi availability, WiFi signal strength, and GPS availability). Although we focus in our implementation on these six contexts, the framework itself is generic and can be extended to much more contexts.

Note that the **item** element can appear multiple times under the same method. This allows the developer to describe complex behaviors. Moreover, the proposed framework is able to resolve, on run-time, conflicts between multiple specifications of sensitivity lists.

The **item** element contains three sub-elements which are **itemName**, **vstart**, and **vend**. The **itemName** represents the name of the context and can take value of any of the pre-described contexts. The **vstart**, and **vend** specifies the value of the specified context for which this method needs to be executed. The ranges for **vstart**, and **vend** vary depending on the **itemName** as follows:

- "BatteryCapacity": The **vstart** and **vend** can take values in the range number between 0% (minimum battery capacity) and 100% (maximum battery capacity).
- "Temperature": Battery temperature can vary based on the environment and the processor usage (which leads to higher current consumption and therefore increase in the battery temperature). The Lithium-Ion battery typically works in the range of -24 degree Celsius up to 70 degree Celsius. Therefore, the **vstart** and **vend** elements can be assigned values between -24 and 70.
- "Voltage": Battery voltage changes across the lifetime of the phone. Most Lithium-ion batteries operate at 3.7v/4.2v the battery starts at 4.2v and falls quickly to about 3.7v for most of the life time of the battery. The **vstart** and **vend** elements can be assigned accordingly.
- "Wifi Availability": The **vstart** can be assigned a binary value of 0 or 1 to reflect whether this method is sensitive to Wifi availability. The value 1 represents the need of Wifi signal for this method to be triggered while the value of 0 represents that the Wifi needs to be absent for this method to operate.
- "Received Signal Strength Indicator (RSSI)": In this case, the The **vstart** and **vend** can be assigned an integer value between 1 and 4 (with 1 being the lowest signal strength). It determines the strength of the received signal. It is calculated based on the proximity relative to the nearest access point.

- "Link Quality": The **vstart** can be assigned an absolute value that defines the link quality in terms of contention, frame rate failure, and interferences. It is different from the signal strength, for example, you may have a high RSSI if you are near an access point. However, the quality of the signal may be poor, if you have high source of interference (e.g. a microwave oven).

A final note is that the proposed framework uses the method that is invoked in the app code as the default method unless the underlined context or operating point changed before the first call of this method.

3.2 Example Of Concrete Configuration File

Figure 3.1 shows a snippet of a configuration file. Methods `solve_CG` and `solve_ChDec` are bound together as alternative implementations for the tag `solve`. These methods are alternative ways to solve set of linear equations. An iterative and approximate method is using Conjugate Gradient, `solve_CG`, while an accurate one is by using Cholesky Decomposition, `solve_ChDec`. Both methods are sensitive to temperature and battery capacity. The first method defines a range of operation of $0 - 60$°C and $21 - 100\%$ remaining battery capacity. The second method can operate in the ranges of $0 - 100$°C and $0 - 100\%$ battery. When the ranges overlap, the method with the lowest priority value (indicating higher priority) is preferred.

3.3 Packing/Unpacking the Configuration File

The configuration file is provided as an Android asset file. This enforces the Android tool chain to pack the configuration file inside the Android application package file (APK).

The APK file is then unzipped during the installation process of the app. The Android operating system generates an app specific cache folder for the purpose of the

```
<Method>
  <MethodName>solve_CG</MethodName>
  <priority>1</priority>
  <tag>solve</tag>
  <item>
    <itemName>battery</itemName>
    <vstart>21</vstart>
    <vend>100</vend>
  </item>
  <item>
    <itemName>temperature</itemName>
    <vstart>0</vstart>
    <vend>60</vend>
  </item>
</Method>
<Method>
  <MethodName>solve_ChDec</MethodName>
  <priority>2</priority>
  <tag>solve</tag>
  <item>
    <itemName>battery</itemName>
    <vstart>0</vstart>
    <vend>100</vend>
  </item>
  <item>
    <itemName>temperature</itemName>
    <vstart>0</vstart>
    <vend>100</vend>
  </item>
</Method>
```

Figure 3.1: Snippet of a CAreDroid configuration file.

installation. At this point, our framework intercepts the normal procedure of the installation, and unpack the configuration file into the cache folder as well. Since unzipping the configuration file and saving it in the cache folder take place only at installation time, the overhead accompanied with these actions does not contribute in calculating the overhead in execution time later in the results section.

written in the asset folder of the application so it is packed inside the Android application package file (APK) with its same format and is installed on the device with the other contents of the APK file. APK files are ZIP file formatted packages based on the JAR file format. This formate facilitates getting the configuration file entry inside the APK and only this entry is unzipped, and not the whole zipped file.

The unzipping of the configuration file entry happens during installation of the application. We use the cache folder created for the installed application to save the configuration file inside it. Having the configuration file written as an asset file and not passing it to the SD-card gives us the ability to expand the size of the configuration file and not be limited by the allowed SD-card storage size. Moreover it provides more modularity by having a cache folder per application rather than having multiple configuration files in one SD-card folder. Since unzipping the configuration file and saving it in the cache folder take place only at installation time, the overhead accompanied with these actions does not contribute in calculating the overhead in execution time later in the results section.

As mentioned, the configuration file is provided by the developer of the application, the reason for that is the developer of the application is the one who provides the alternatives of the sensitivity methods and hence the assumption here is that the developer is aware of the operation ranges for each sensitive methods that is defined in the application. We are aware that in [need citation here], some tools can figure some parts of the software that are candidates for optimizations but most of these tools take into account the execution time optimization. Our framework works with more than one optimization parameter to choose the best method in the sense of best effort that fits the current operation point.

CHAPTER 4

CAreDroid Optimization Process

4.1 The Optimization Process

During the installation of the APK file, the Android flow creates a new Dalvik Virtual Machine (DVM) to host this application [1]. During this process, the Android flow extracts the Dalvik Executable Files (DEX) which contain the compiled byte codes. It starts by building specific data structures that captures all the information about the classes and the methods that is presented inside the DEX file.

After the application is installed, and during the first execution of the application, the Android flow performs several optimization procedures and initializations on both the DEX file as well as on the extracted data structures. The final output is then saved in the Optimized Dalvik Executable Files (ODEX) and its corresponding data structures, which are loaded in the memory of the virtual machine.

The final ODEX data structures captures the offset of the byte code that corresponds to each method. Therefore, whenever a method is called, the Dalvik virtual machine accesses this data structure in order to jump to the correct address of the byte code.

Our CAreDroid framework aims mainly to switch between context-based polymorphic methods on run time. Hence, it is important to be able to capture the information about the offsets, in the DEX byte code, of the context-based polymorphic methods. Therefore, the CAreDroid flow intercepts the process of building the DEX data structure and the corresponding ODEX data structures in order to configure and optimize

[1] Android security mechanisms rely mainly on the concept of sandboxes. That's why each application is hosted on a separate virtual machine.

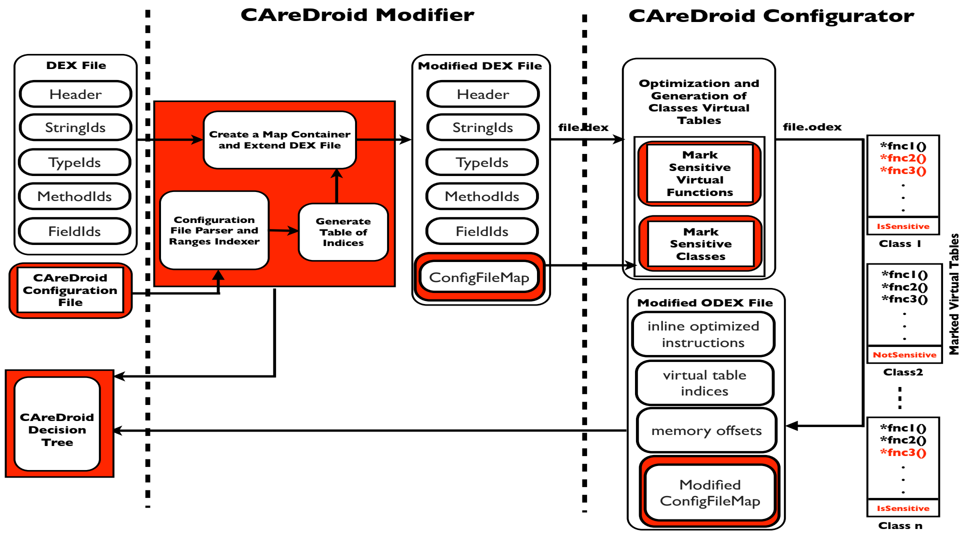


Figure 4.1: CAreDroid Optimization Process

the CAreDroid runtime engine.

The optimization process consists of two main components; CAreDroid DEX Modifer and CAreDroid Configurator (Figure 2.2). Figure 4.1 shows the architecture of these two components. The final output of this process is two *map* data structures called *Replacement Map* and *Table Of Indices*. This section is devoted to discuss the details of this optimization and configuration process.

4.2 The *Replacement Map* (RM) and the *Table Of Indices* (TOI)

As shown in Figure 4.1, the *Replacement Map* and the *Table Of Indices* are the final two outputs of the configuration process. During this process, the values stored inside these data structures is edited several times till they hold the final data. In this subsection, we introduce these two data structures and point their importance.

To motivate for the importance of these two data structures, we examine how the Java code is compiled into DEX and then optimized into ODEX. We consider the example shown in Listing 4.1 - Listing 4.3. The Java code in Listing 4.1 instantiates an object from the class "MyClass" and then calls the method called "foo".

The corresponding DEX code is shown in Listing 4.2. The DEX compiler assigns a unique ID number for each class. For example, the “MyClass” is assigned the id “type@0225”. Similarly, the methods are also assigned for a unique ID. For example, the default constructor of the “MyClass” is assigned the ID of “method@101” while the “foo” method is assigned to the ID “method@101e”.

On the other side, all objects are renamed by the DEX compiler. Unlike the class names and the method names, the new object names are used in the DEX code and the original name completely disappears. For example, the “myObject” instance is renamed to “v14”.

The method call is done through the the special byte-code named “invoke-virtual” which takes the reference to the object (“v14” in our case) as its argument. Nevertheless, the original name of the method is still used to call the methods. The method ID is used internally to point to the offset, in the DEX code, that points to the implementation of that method.

Once the DEX file is optimized and the ODEX file is generated, the method names and the IDs disappears. Instead, the index inside the virtual-table is used to access the code. Note that the Java method name and the DEX method id are no more usable once the ODEX is generated. The three steps of tracking the methodID and classID is shown in Table 4.1

Our framework aims to switch between different methods by hacking the virtual table and pointing to the method that best fits the current context. Therefore, its is of great importance to track how the Java methods are assigned to IDs and then how these IDs are translated to virtual table indices. This is the main role of the TOI and RM data structures.

Table 4.1: Replacement Map table. Method ID is updated at the three steps of generating the ODEX file

ClassID	MethodID
...	...
256	~~4126~~ (Step 1) ~~5~~ (Step 2) 11 (Step 3)
256	~~4127~~ (Step 1) ~~6~~ (Step 2) 12 (Step 3)
256	~~4128~~ (Step 1) ~~7~~ (Step 2) 13 (Step 3)
...	...

Listing 4.1: Java File

```
MyClass myObject = new MyClass();
myObject.foo();
```

Listing 4.2: classes.dex file

```
new-instance v14, Lcom/Application_Name/MyClass;
// type@0225

invoke-direct {v14}, Lcom/Application_Name/MyClass;.<init>:([[D)V
// method@101

invoke-virtual {v14}, Lcom/Application_Name/MyClass;.foo;
// method@101e
```

Listing 4.3: invoke_virtual format in ODEX

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

4.2.1 The *Table Of Indices* (TOI)

The CAreDroid runtime engine checks the current context of the phone. It then picks the method which best suits the current context and switch, transparently, to this method.

Note that the developer can, in general, specify conflicting switching operation ranges. That is, different polymorphic methods are sensitive for the same range of values for a specific context. Therefore, the CAreDroid have a runtime, conflict resolution mechanism. The TOI serves as the main data structure to discover these conflicts.

The TOI consists of multiple *associative arrays*. For each of the six contexts (battery capacity, temperature, voltage, ... etc), we create a corresponding associative array. Each associative array is initialized with the *default* operation range which is the full operation range that this context can take. Each operation range is assigned a unique identifier, called *operation range identifier*. The unique *operation range identifier* for the *default* operation range is assigned to zero.

For each sensitive method, we check the specified sensitivity list and the associated operation range. For each operation range specified in the XML configuration file, the corresponding context-associative array stores this range versus a generated unique *operation range identifier*.

On runtime, the CAreDroid runtime engine utilizes the TOI along with the current phone context to retrieve all the *operation range identifiers* which is satisfied by the current phone context. If there exist more *operation range identifier*, except for the default operation range, a conflict is discovered. The CAreDroid runtime engine then utilizes a decision tree in other to resolve this conflict (Chapter 5).

4.2.2 The *Replacement Map* (RM)

The replacement map[2] is a *map* data structure.The *key* of this map is a composite key which consists of the pair of class-id and method-id. The *value* field, of this map, is an array whose length is equal to the number of contexts (six in the current implementation). This array specifies the operation range for this method for all the six contexts. This association is done by copying the corresponding *operation range identifier* from

[2]Since each class can have multiple sensitive classes, our implementation creates a separate RM for each sensitive class. However, for clarity of discussion, we refer to the RM as a single data structure.

Class ID	Method ID	B	T	V	W	S	Q
class 1	method 1	0	0	0	0	0	0
class 1	method 2	1	1	0	0	0	0
class 2	method 1	2	2	1	0	0	0
class 2	method 2	2	2	1	0	0	0

Table 4.2: Replacement Map. B: Battery Capacity, T: Battery Temperature, V: Battery Voltage, W: WiFi connectivity, S: Signal strength, Q: Signal Quality

the TOI data structure. An example of the RM is shown in Table 4.2.

4.3 CAreDroid DEX Modifier

The values of the TOI and the RM are filled by the CAreDroid DEX Modifier and the values of the RM is changed to reflect the changes done by the Android DEX Optimizer on the method ids and offsets.

The CAreDroid DEX Modifier is executed within the installation process of the APK file. It is issued just after the Android flow has extracted the DEX data structure. It is responsible for building the TOI and RM and modify the internal data structures accordingly.

Figure 4.2 shows the structure of the DEX data structures. The current data structure, contains a set of arrays that holds the names and the IDs of all classes, methods, fields, and method prototypes inside these classes. We modified the DEX data structure by adding two fields. These two fields are pointers to the generated RM and the TOI data structures.

First, The TOI is constructed by parsing the XML configuration file. This process takes place just after the installation of the APK file. The TOI is then added to the DEX data structure.

Second, the RM is constructed based on the information in the DEX data structure and the TOI. This is done by matching the *ClassName* and the *MethodName* elements in the XML file with the strings in the *MethodIDs* in the DEX data structure.

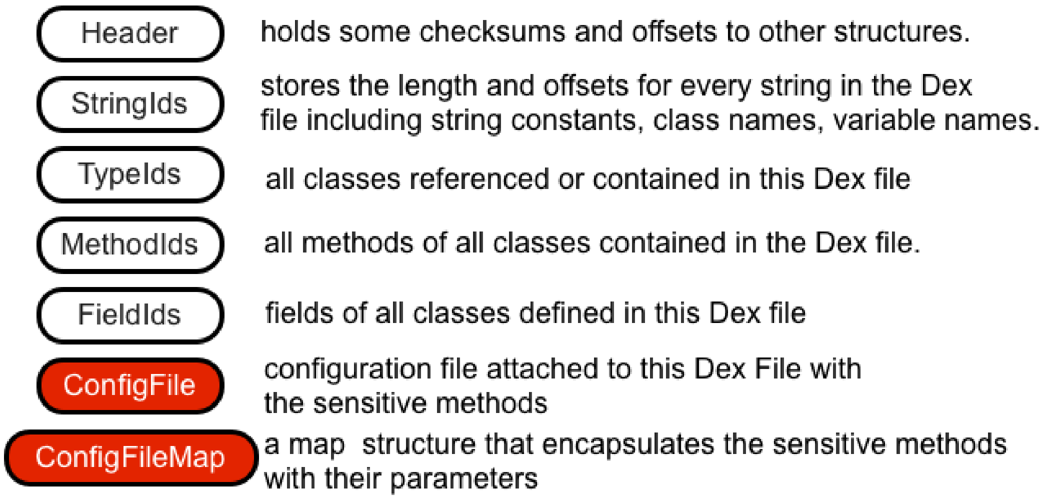

Header
holds some checksums and offsets to other structures.
StringIds
stores the length and offsets for every string in the Dex file including string constants, class names, variable names.
TypeIds
all classes referenced or contained in this Dex file
MethodIds
all methods of all classes contained in the Dex file.
FieldIds
fields of all classes defined in this Dex file
ConfigFile
configuration file attached to this Dex File with the sensitive methods
ConfigFileMap
a map structure that encapsulates the sensitive methods with their parameters

Figure 4.2: The fields of the extended DEX data structure. The unmarked fields represents the original fields in the DEX data structure while the marked fields represents the fields added by the CAreDroid DEX Modifier.

Note that all these string information are not available once the DEX file is processed by the Android flow and the Optimized DEX (ODEX) file is generated. Therefore, it is important to intercept the Android flow at the installation time in order to utilize these information along with the information in the configuration XML file. In the next subsection, we examine how to modify the data inside the RM to reflect the changes made by the Android DEX optimizer.

4.4 CAreDroid Configurator

After the DEX Modifier constructs the TOI and the RM data structures, the DEX file passes through the normal Android optimization process, resulting into the generation of the ODEX file. During this process, a virtual table, for each class, is generated and methods are assigned for different method IDs that fits the ODEX structure.

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 CAreDroid framework intercept this process and modify the RM data structure accordingly.

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 being resolved. In particular, a virtual table is generated for each class. Each resolved method correspond to an entry in this virtual table. Therefore, each method is now identified with its unique index inside the virtual table. For example, to call a method, the ODEX utilizes the “invoke-virtual-quick” byte code along with the method offset in order to correctly call a method (Listing 4.3). As before, the CAreDroid framework intercepts this process and edits the values inside the RM accordingly.

Finally, we extend the ODEX file structure by adding two references to the TOI and the RM data structures which are generated by the prescribed process. We also extended the internal Android *object* and *method* data structures by adding *sensitivity flags*. These flags are used later by the Runtime Engine to facilitate the method switching.

CHAPTER 5

CAreDriod Runtime Engine

5.1 Overview of CAreDriod Runtime Engine

Once the ODEX file is generated, the application is now ready to run over the Dalvik Virtual Machine (DVM) runtime. The Dalvik interpreter executes the instructions saved in the ODEX file. DVM runtime utilizes two types of interpreters. The first is called the "portable" interpreter which is implemented in C language and is able to work on any platform. The second interpreter is called the "fast" interpreter which is implemented in assembly and tailored towards specific platforms. The DVM supports switching between the two interpreters on run-time.

In our framework, we extend the "portable" interpreter to support the CAreDriod runtime engine. The extended interpreter checks the current interpreted ODEX OpCode. Whenever the OpCode corresponds to the "invoke-virtual" instruction (see Listing 4.3), the extended interpreter executes the CAreDriod runtime engine.

The CAreDriod runtime engine checks the arguments of the "invoke-virtual" instruction— the method ID and the class ID — against the *sensitivity flags* (previously introduced in Chapter 4 as an extension to the Android *method* data structure). If the *sensitivity flag* is set, then CAreDriod runtime engine has encountered a sensitive method.

For sensitive methods, the CAreDriod runtime engine examines the current platform context (also called the current operating point). This is done by snooping on the underlying Linux device drivers. Once the operating point is determined, the CAreDriod runtime engine picks the method that fits the current operating point based on the con-

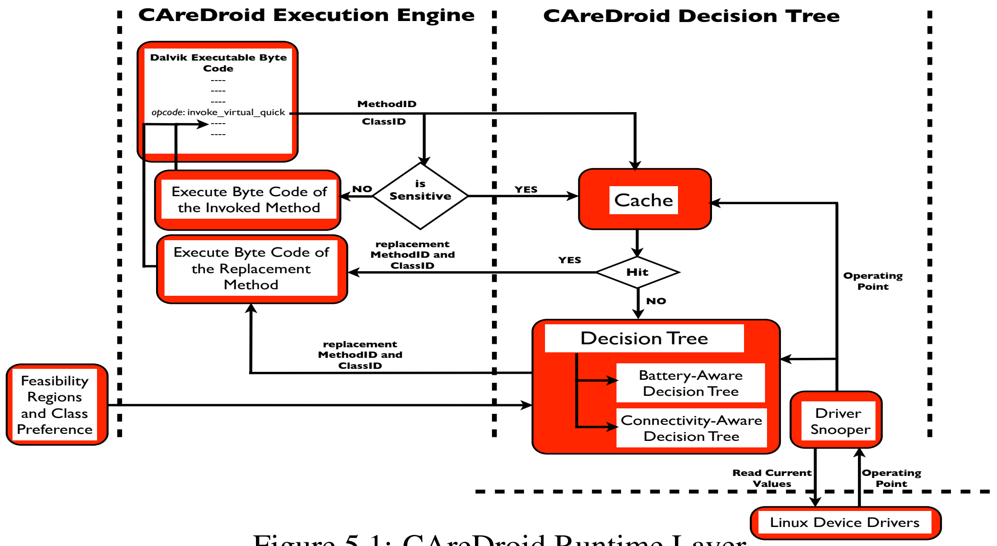

CAreDroid Execution Engine
CAreDroid Decision Tree
Dalvik Executable Byte Code
opcode: invoke_virtual_quick
MethodID
ClassID
is Sensitive
NO
YES
Execute Byte Code of the Invoked Method
Execute Byte Code of the Replacement Method
replacement MethodID and ClassID
Cache
Hit
YES
NO
Operating Point
Decision Tree
Battery-Aware Decision Tree
Connectivity-Aware Decision Tree
replacement MethodID and ClassID
Feasibility Regions and Class Preference
Driver Snooper
Read Current Values
Operating Point
Linux Device Drivers

Figure 5.1: CAreDroid Runtime Layer

figuration delivered by the user and captured in the RM data structure. This process requires to resolve the conflicts in the user configuration. The CAreDriod runtime engine utilizes the TOI data structure to execute the selected method instead of the original method. This process is illustrated in Figure 5.1.

The CAreDriod runtime engine consists of three blocks. The first block is named the “Driver Snooping” block. This block is responsible to obtain the current operating point. The second block is called the “decision tree”, which is responsible for conflict resolution and selecting the method that fits the operating point. Finally, in order to optimize the execution of the CAreDriod runtime engine, the third block, “cache”, is used to cache the conflicts resolved recently in order to by pass unnecessary usage of the “decision tree”. This section illustrates the details of each of these blocks.

5.2 Driver Snooper

Android system leverages the information about the current context to the software stack through the Hardware Abstraction Layer (HAL). The HAL consists of a set of sensor managers that work as an intermediate layer between the low level drivers and the high-level Java applications.

In our framework we snoop on the interface between the HAL and the low level device drivers. This interface is called the “sysfs” virtual file system. Each device driver exports his data into a set of files located under “/sys/class/ ”. By reading these files, our framework is able to determine the state of the battery, WiFi and GPS. This state is then sent to the decision tree in order to pick the right method.

5.3 CAreDroid Decision Tree

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. We use the information presented in the RM data structure (associated with this sensitive method) to get all *candidate methods*. Since a conflicting configuration file may result into multiple candidate methods, we need to resolve this conflict and choose one method out of this set of *candidate methods*.

For each *candidate method*, the RM data structure associates six — one for each context — *operation ranges*. We differentiate between two cases. In the first case, a method is considered a *candidate method* if the operating point satisfies *all* the six *operation ranges*. On the other hand, the second case considers all methods for which the operating point satisfies *at least one* of the *operation ranges*. Therefore, our framework implements two conflict resolution policies, called “must fit” and “best fit”.

In the “must fit” policy, conflicts are resolved based on the rank associated to each method (defined by the **orderOfPreference** element in the Configuration XML file).

In the “best fit” policy, the framework utilizes both the method rank as well as the preference for the context parameters (defined by the **Preference** element in the Configuration XML file). The framework resolves the conflicts first by resorting to the context preferences, that is, reducing the set of the *candidate methods*, based on whether they satisfy the highest preferred context and so on. Further conflicts are resolved using the method rank.

Choosing which method to execute depends on the best method in the feasible sets in the sense of meeting the user *preference* as stated in the configuration file. If the *preference* is a Power preference then the choice is made based on this order: the battery capacity requirement first then meeting the requirement for battery temperature and finally meeting the requirement of the battery voltage for the battery related parameters then considering the connectivity parameter. However, if the *preference* is a Connectivity preference then the wifi status parameter is considered first and then the battery related parameters are taken into account in the same order as in the Power preference.

If none of the available feasible sets intersect with the operating point, the decision tree will output the *replacement method* to execute based on its order of preference stated in the configuration file.

The feasible set for the battery related parameters divide the space into 3-dimensions in which each of the three battery parameters create a plane in its dimension and the intersection of three planes defined by the ranges on each dimension gives a possible candidate of a method to execute. If there is no intersection between three planes, the candidate method is chosen from the possible intersection of two planes. If the there are two possible candidates for 2-planes intersections, the candidate method is chosen from the planes of more importance to meet as stated by the *preference*.

Since our design select the battery capacity to be the most important metric in the battery related parameters then if for example the are two 2-planes that are formed, one with battery capacity plane and battery temperature plane and the other with battery temperature plane and battery voltage plane, the candidate method is chosen from the feasibility set of the former 2-planes intersection (battery capacity and battery temperature).

As stated above the choice of the candidate method is based on best effort. First, the algorithm tries to find a feasibility set with three intersecting planes (battery capacity, battery temperature, battery voltage), if not found, it tries to allocate feasible sets with two intersecting planes and make a preference choice on them based on the relative importance of the intersecting planes. However, if the later step does not succeed meaning that the operating point does not lie in any intersecting planes, it tries to allocate isolated planes. The algorithm then chooses the planes with relative importance as defined above. If all the previous steps fail then the algorithm will return a method based on its order of preference value in the configuration file.

For example if an operating point lie in an isolated plane of battery capacity and there are two candidate methods defined in this feasible set then the algorithm will return the one with the higher order of preference.

The different types of candidate sets and the decision of the replacement method from them is shown in Table 5.1. In this table, **Level A** is the highest level in which the operating point meets the three constraints defined by the three battery parameters for more than one method, in this case the decision is based on the order of preference.

Level B Type1 is the case when the operating point satisfies only two constraints in more than one one methods but they are the same two constraints, in this case the decision is based on the order of preference.

Level B Type2 is the case when the operating point satisfies two different constraints in more than one method, in this case the decision is taken based on the priority of the constraints.

Level C Type1 happens when the operating point satisfies one constraint in more than one method but it the same parameter associated with this constraint and in this case the decision is based on the order of preference of the methods involved.

Level C Type2 is the case when the operating point satisfies one constraint but for different parameters in more than one method, and in this case the decision is based on the priority of the parameters.

At run-time an arbitrary operating point can be located on a plane of different intersection polygons and the determination of which intersection polygon as explained above is used to select the candidate methods. An example for an arbitrary point is shown in Figure 5.3 where three feasibility regions for three sensitivity methods along with four different arbitrary operating points.

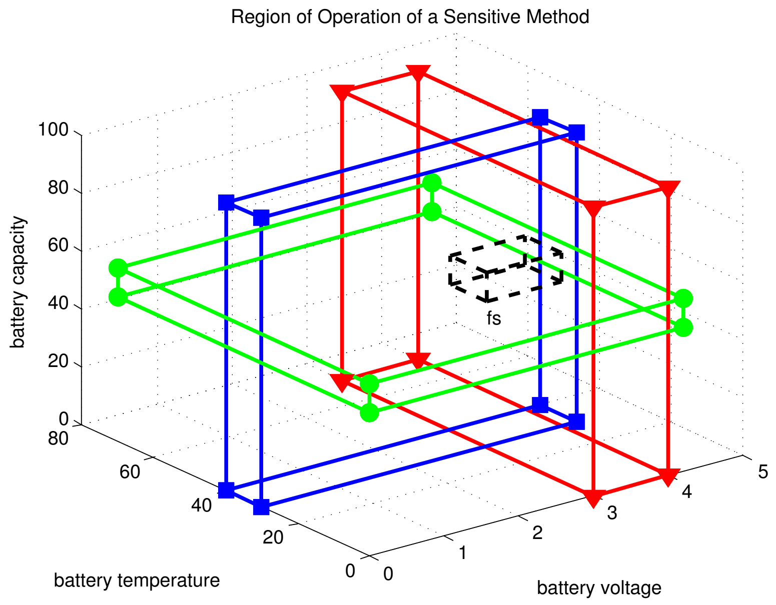
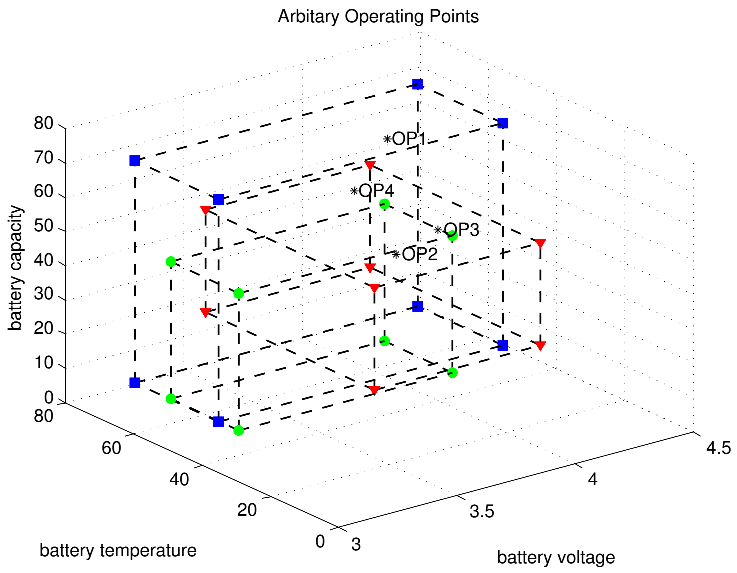


Figure 5.2: Region of Operation for one sensitive method f_s with arbitrary values; The intersection of the three regions defined by the polygons marked in ▼ and ■ and •; which represent the regions of battery voltage, and battery temperature, and battery capacity respectively; is the polygon drawn with the dashed lines which represents the feasibility region for this method

Table 5.1: Candidate Sets Levels

Candidate Set Level	Candidate Set Property	Decision from the Candidate Set
LevelA	Satisfies the three constraints	Based on the higher order of preference
LevelB Type1	Satisfies similar two constraints	Based on the higher order of preference
LevelB Type2	Satisfies different two constraints	Based on the priority of parameters
LevelC Type1	Satisfies similar one constraint	Based on the higher order of preference
LevelC Type2	Satisfies different one constraint	Based on the priority of the parameters

Figure 5.3: Arbitary Operating Points (OP) with respect to three sensitivity methods; f_{s1}, and f_{s2}, and f_{s3} are marked by ●, and ▼, and ■ respectively.
OP1: Candidate Set Level-A = $\{f_{s3}\}$;
Candidate Set Level-A = $\{f_{s1}, f_{s2}, f_{s3}\}$;
Candidate Set Level-A =$\{f_{s2}, f_{s3}\}$;
Candidate Set Level-A = $\{\}$;
Set Level-B Type 2 = $\{f_{s1}, f_{s2}\}$.



5.4 Optimization Cache

In order to decrease the overhead of the conflict resolution mechanism, our framework utilizes a "conflict resolution cache". Similar to the concept of "temporal locality", our heuristic assumes that the operating point does not change in small scale granularities. Therefore, if a method is called multiple times within a short amount of time (inside a loop for example), the same polymorphic implementation will be used for all these multiple calls.

The designed cache is used to store the recently resolved conflicts, that is, the recent operating points along with the method that fits this operating point. Each entry corresponds to the six value of the operating point along with the method ID for the selected method. The cache utilizes a Least Recently Used (LRU) approach to replace existing entries with newer enteries.

CHAPTER 6

Evaluation

Evaluation of CAreDroid is carried out on two phases. The first phase, is characterization of the CAreDroid overhead and comparison between context-aware platforms versus non context-aware ones. In the second phase, we present a complete app that can benefit from the CAreDroid framework. In these results, our framework gives an order of 3–12x saving in execution time in comparison to context-aware platforms that do not rely on CAreDroid.

In both phases, we focus on arithmetic/signal processing test cases with variable output quality. This is motivated by the work on approximate computing [KK12, KGE11, Tis09, ACN11]. Approximate computing utilizes variable quality in order to double the efficiency and reduce energy consumption [HO13]. Using CAreDroid, approximate computing, now, can be implemented to be context-aware. That is, different accuracy levels can be attained based on the underlying context of the phone.

The Java layer test cases are carried over a Nexus 4 phone running a modified system image for platform 4.2 API 17 [Anda]. The execution time is obtained using the Android SDK tracer [Andb]. The native layer test cases are carried over VarEMU Virtual Machine Monitor [WEL13]. VarEMU facilitates obtaining detailed energy measurements on method granularity.

6.1 Case Study: Variable quality arithmetic

In this section we present a series of small numerical applications that serve as microbenchmarks to demonstrate the benefits and overheads of CAreDroid.

6.1.1 CAreDroid/Java: Linear Equations Solver

In this example, we use three linear equations' solvers, named LUP-decomposition (LU), Cholesky decomposition (CHD), and Conjugate Gradient (CG). These three methods have different memory and convergence time characteristics. We implemented a simple application that solves 20 linear equations in 20 unknowns using the three mentioned matrix solvers. In order to characterize the CAreDroid overhead, we generated an arbitrary configuration file which assigns each of the three solvers to different battery and connectivity contexts. We evaluated our 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 SDK.

Table 6.1 shows the CPU time for each individual solver if it is used in a non context-aware system. It also shows the overhead for both the pure Java implementation and CAreDroid calculated as the percentage increase of the CPU time compared to the non context-aware implementation. Table 6.1 shows that CAreDroid outperforms the pure Java implementation with around 3–12x savings in CPU time while adding a minimal overhead (2.5%–5%) compared to the non context-aware case. The results in Table 6.1 also shows that the driver shopper mechanism used by CAreDroid is a major factor of reducing the overhead compared to the HAL interface provided by the Android flow. The results also shows that *best fit* policy adds a slightly more overhead compared to the *must fit* policy due to the complexity of the decision tree 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. The execution time and overhead comparison are shown in Figure 6.1.1 and Figure 6.1.1 respectively.

Platform	Solver	CPU time[a] (ms)	Overhead
Non-context aware (Base)	LU	8.322 + 0	-
	CHD	16.872 + 0	-
	CG	13.375 + 0	-
Context aware (Pure Java)	LU	8.549 + 2.56	32.6%
	CHD	17.648 + 2.56	18.9%
	CG	13.736 + 2.56	21.26%
CAreDroid (Must Fit)	LU	8.54 + 0	2.55%
	CHD	17.277 + 0	2.34%
	CG	13.867 + 0	3.55%
CAreDroid (Best Fit)	LU	8.748 + 0	4.87%
	CHD	17.734 + 0	4.86%
	CG	13.882 + 0	3.65%

[a]CPU Time = Method time + HAL Callback time

Table 6.1: Results of the execution time for the different implementations of the linear equations solver example.

Platform	CLOC	Overhead
Non-context aware (Base)	275	-
Context-aware (Pure Java)	606	54.62%
CAreDroid	275 +156[a]	36.19%

[a]XML Configuration file

Table 6.2: Results of the Count line of code (CLOC) for the different implementations of the linear equations solver example.

Table 6.2 shows the total number of line of code (CLOC) for each of the implementations. It is direct to observe the reduction in CLOC for the applications written using CAreDroid compared to the pure Java implementation for this particular example.

6.2 Case study: A Context-Aware Image Processing App

Various social media apps provide the functionality of modifying/enhancing pictures before sharing them. This functionality rely on different image processing primitives. One of the most important image processing primitives is edge detection which is heavily used in feature detection/extraction and face recognition [Yak76, YH94]. We focus

Figure 6.1: Execution Time Comparison for Matrix Solver Example

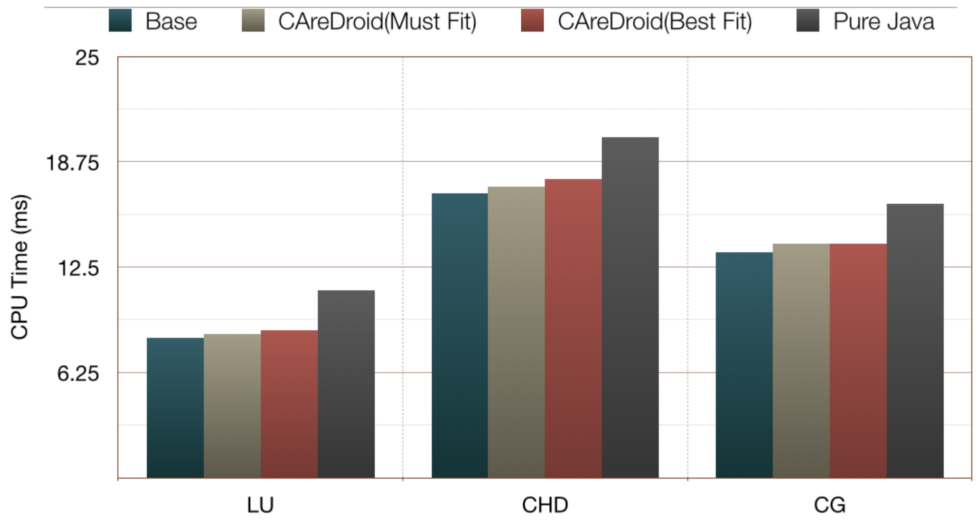
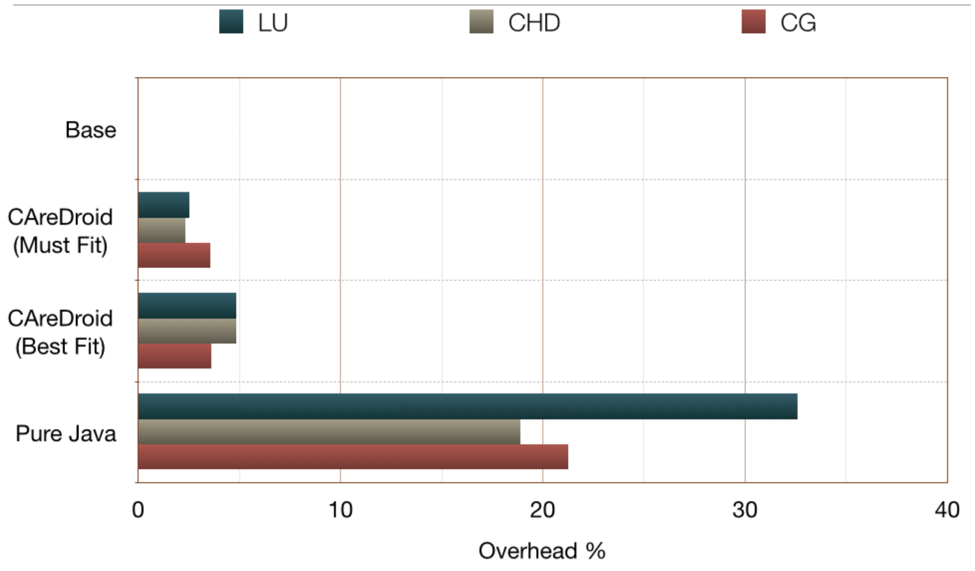


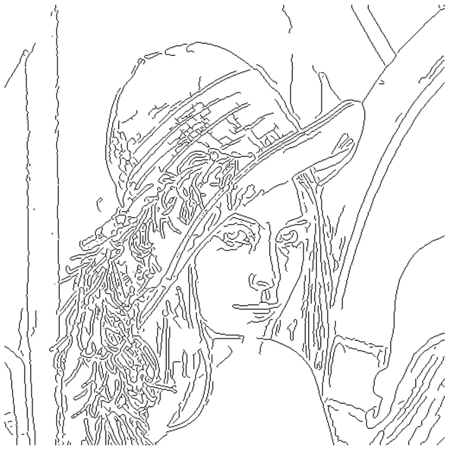
Figure 6.2: CAreDroid Execution Time Overhead for Matrix Solver Example

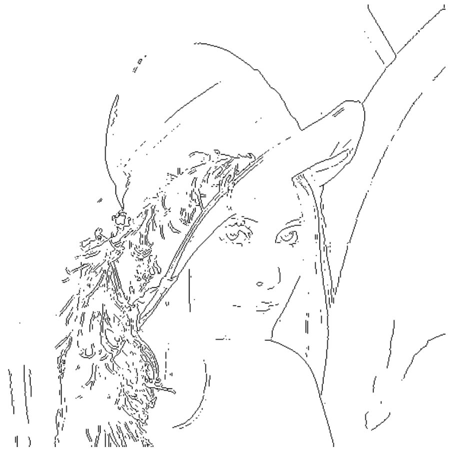


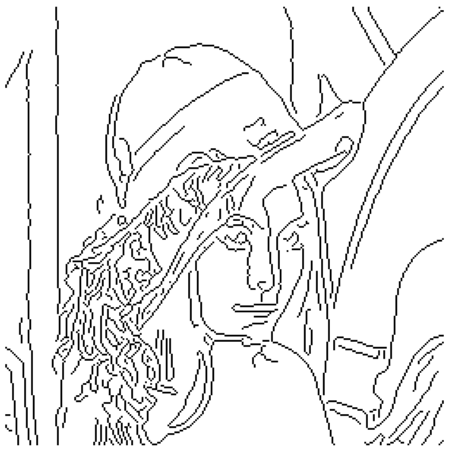
this study on performing edge detection with variable quality based on the underlying context.

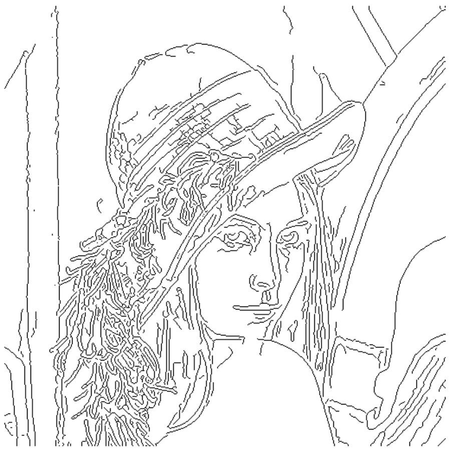
The app utilizes different polymorphic implementations of the edge detection described as follows:

1. **Default**: This implementation uses *Canny filter* [Can86] which gives the highest


(a) Default


(b) Optimized


(c) Fast


(d) Server-Client

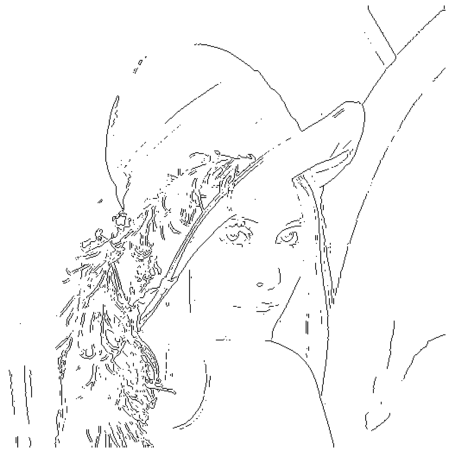

(e) Opt. Server-Client

Figure 6.3: Accuracy results of the different polymorphic methods used for edge detection.

accuracy and runs locally on the mobile. This is the default implementation in the normal context.

2. **Approximate**: When the battery capacity is critical (very low), it is important to reduce the amount of computations. Hence, this implementation use *Sobel Filter* which is less computationally intensive [MA09].

3. **Fast**: High increase in the battery temperature is an indication of having multiple apps running. Therefore, in order to enhance the response of the app, this implementation first down scales the image size then performs a *Canny filter* on the reduced size image leading to significant decrease in computation time leaving more room for the rest of the apps.

4. **Server-Client**: If the WiFi connectivity is available with good quality, the app can remove the burden of enhancing the image to a remote server. In this implementation the computationally expensive *Canny filter* is implemented on a remote server and the results are pushed back to the app.

5. **Approximate Server-Client**: This implementation is used when the battery capacity is approaching critical value (very low) and the WiFi is available with good quality. This implementation utilizes a *Sobel filter* implemented on a remote server. By using the *Sobel filter*, the server will be able to push back the image to the app without leaving the app waiting for longer time to accommodate with the low capacity in the battery.

The accuracy result of the five edge detection method is shown in Figure 6.3. To better understand these accuracies, we compare all implementations to the *Canny filter*. We define the number of false positives as the number of pixels that the algorithm detects as an edge falsely compared to the edges detected by the *Canny filter*. Similarly, we define the number of false negatives as the number of pixels that the algorithm fails to detect as an edge. Figure 6.4 shows the percentages of false positives and false neg-

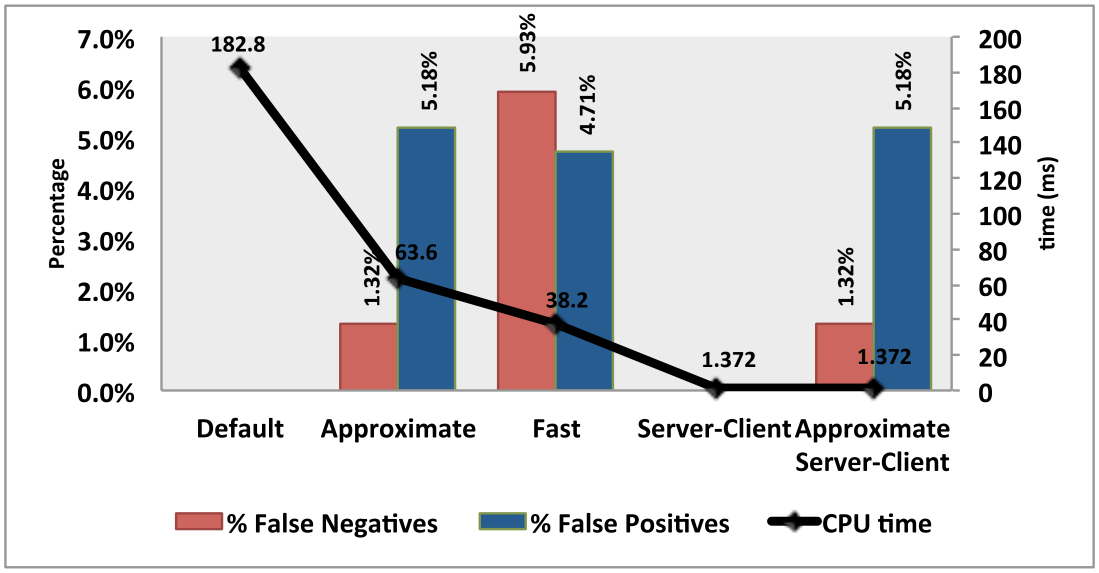


Figure 6.4: Edge Detection Results. Percentage of false positives and false negatives versus CPU execution time.

atives with the execution time for the five edge detection methods. These results reveal the tradeoff between different methods in terms of accuracy versus execution time. it is obvious to note that server-client implementations requires negligible execution time since all the computation is done on the server. It is also observed that the execution time of the *fast* method is reduced to nearly quarter the time of the default, which was expected because we scaled the image to quarter the size in this example.

In this case study, we show that context awareness can be manifested to show the tradeoff between execution time and accuracy.

CHAPTER 7

Discussion

In this section we discuss some aspects of the CAreDroid in terms of assumptions, limitations, and portability.

7.1 Limitations and Considerations

Assumptions: CAreDroid assumes that the app developer is aware of the suitable ranges of operations for the different *sensitive* methods.

Limitations: In CAreDroid/Java layer, the polymorphic methods should be pure functions. Pure functions must satisfy three conditions:

1. Does not depend on I/O devices or external input.
2. Does not change any global state that may change how the program behaves or proceeds in the middle of the execution.
3. The output result depends only on the function arguments. In other words, the function evaluates the same result given the same arguments.

However, in CAreDroid/Native layer, functions need not be pure, but they must have consistent side effects and state—i.e., each alternative implementation of a function must manipulate any global state consistently and atomically for each function call.

This design choice simplifies function mutation both in implementation and in runtime complexity.

7.2 Portability

CArDroid framework is a firmware modifier. The framework creates a new system image that has to be flashed on the Android platforms for our system to work. The framework is based on Android Open Source Project (AOSP)-API level 17. CArDroid execution engine is an extension to the Android portable interpreter. Hence, it can be used on any Android platform and is not architectural dependent.

7.3 Enhancements

Portable interpreter has a negative effect on the execution time of the app, albeit it makes the framework portable. To address this issue, we enable the possibility of the switch between fast interpreter and portable interpreter to happen at run-time. The execution will start normally using the Just-In-Time compiler (JIT) as a "fast" interpreter and when the interpreter hits a calling from a sensitive class, the interpreter will switch to the "portable" interpreter. After executing the sensitive method the interpreter will switch back to the "fast" version. This enables our framework to work on different platforms and at the same time benefit from the fast interpreter.

CHAPTER 8

Conclusion

Context-aware computing is a powerful technique for adaptation and energy-awareness in mobile systems. It improves resource usage of mobile systems by adapting to context. In this paper we presented CAreDroid, a framework for context-aware computing in Android. CAreDroid allows for applications to be context-aware without having to directly deal with context sensors in 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 files. A configuration file may, for example, indicate that a certain implementation of a method should be used when battery capacity is low, or when WiFi signal quality is good. In this paper we showed how context-awareness can help applications reduce their resource usage in constrained contexts, or alternatively provide high-quality results in contexts where resources are abundant. Results with variable quality arithmetic operations show how CAreDroid can provide a band of upwards of 40% in energy savings at the cost of upwards of 0.6% degradation in application output quality. We further demonstrated how CAreDroid can provide context-awareness more efficiently and in fewer lines of code than application-specific context-aware implementations.

Polymorphic methods in CAreDroid must be pure functions, i.e., they cannot perform I/O or cannot change global program states, and their output must depend only on the function arguments. To allow for non-pure functions, the framework would require state migration procedures between every possible pair of alternate functions. Function mutations would also be more costly in runtime—our simpler design choice requires only replacing the function itself for each migration. While lifting these design restrictions could potentially lead to broader choices of function adaptation, we believe that would lead to an impractical system both in terms of development and runtime overhead.

CAreDroid assumes that an application developer can provide multiple implementations of sensitive methods. We believe that, for some applications, polymorphic system services such as the variable quality math library demonstrated in this paper could provide significant benefits without incurring in additional application implementation complexity. CAreDroid also expects the application programer to be aware of suitable ranges 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. CAreDroid contexts are modular, and we expect to extend available contexts to include other sensors such as battery health, state of charge, or communication link noise. We furthermore intend to abstract context information from raw sensor data into high level inferences. This could allow the programmer, for example, to configure applications to adapt to user activity contexts (such as *driving*, *walking*, or *running*), locations (such as *at home*, *at work*, or *on travel*), emotional status (such as *calm* or *nervous*), or expected charging behavior.

References

[ACN11] P. Albicocco, G.C. Cardarilli, A. Nannarelli, M. Petricca, and M. Re. “Degrading precision arithmetics for low-power FIR implementation.” In *Circuits and Systems (MWSCAS), 2011 IEEE 54th International Midwest Symposium on*, pp. 1–4, Aug 2011.

[ACW09] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. “PetaBricks: a language and compiler for algorithmic choice.” *SIGPLAN Not.*, **44**:38–49, June 2009.

[Anda] Android. “Android Open Source Project.” https://source.android.com.

[Andb] Android SDK . “Profiling with Traceview.” http://developer.android.com/tools/debugging/.

[BC10] Woongki Baek and Trishul M. Chilimbi. “Green: a framework for supporting energy-conscious programming using controlled approximation.” *SIGPLAN Not.*, **45**:198–209, June 2010.

[BGX10] Aaron Beach, Mike Gartrell, Xinyu Xing, Richard Han, Qin Lv, Shivakant Mishra, and Karim Seada. “Fusing Mobile, Sensor, and Social Data to Fully Enable Context-aware Computing.” In *Proceedings of the Eleventh Workshop on Mobile Computing Systems &#38; Applications*, HotMobile ’10, pp. 60–65, New York, NY, USA, 2010. ACM.

[Can86] John Canny. “A Computational Approach to Edge Detection.” *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, **PAMI-8**(6):679–698, Nov 1986.

[FJ05] Matteo Frigo and Steven 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”.

[HO13] Jie Han and Michael Orshansky. “Approximate computing: An emerging paradigm for energy-efficient design.” In *Test Symposium (ETS), 2013 18th IEEE European*, pp. 1–6. IEEE, 2013.

[KGE11] P. Kulkarni, P. Gupta, and M. Ercegovac. “Trading Accuracy for Power with an Underdesigned Multiplier Architecture.” In *VLSI Design (VLSI Design), 2011 24th International Conference on*, pp. 346–351, Jan 2011.

[KK12] A.B. Kahng and Seokhyeong Kang. “Accuracy-configurable adder for approximate arithmetic designs.” In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pp. 820–825, June 2012.

[LGP07] X. Li, M.J. Garzaran, and D. Padua. "Optimizing Sorting with Machine Learning Algorithms." In *Proc. Parallel and Distributed Processing Symposium*, 2007.

[Lie86] Henry Lieberman. "Using Prototypical Objects to Implement Shared Behavior in Object-oriented Systems." *SIGPLAN Not.*, **21**(11):214–223, June 1986.

[LLH13] Ting-Yi Lin, Ting-An Lin, Cheng-Hsin Hsu, and Chung-Ta King. "Context-aware decision engine for mobile cloud offloading." In *Wireless Communications and Networking Conference Workshops (WCNCW), 2013 IEEE*, pp. 111–116, April 2013.

[LMM07] Andreas Lachenmann, Pedro José Marrón, Daniel Minder, and Kurt Rothermel. "Meeting lifetime goals with energy levels." In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pp. 131–144, New York, NY, USA, 2007. ACM.

[MA09] Raman Maini and Himanshu Aggarwal. "Study and comparison of various image edge detection techniques." *International Journal of Image Processing (IJIP)*, **3**(1):1–11, 2009.

[PZC14] 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.

[RSH08] N. Ravi, J. Scott, Lu Han, and L. Iftode. "Context-aware Battery Management for Mobile Phones." In *Pervasive Computing and Communications, 2008. PerCom 2008. Sixth Annual IEEE International Conference on*, pp. 224–233, March 2008.

[RZ11] A. Rahmati and Lin Zhong. "Context-Based Network Estimation for Energy-Efficient Ubiquitous Wireless Connectivity." *Mobile Computing, IEEE Transactions on*, **10**(1):54–66, Jan 2011.

[SKG07] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery 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, pp. 161–174, New York, NY, USA, 2007. ACM.

[ST94] B.N. Schilit and M.M. Theimer. "Disseminating active map information to mobile hosts." *Network, IEEE*, **8**(5):22–32, Sept 1994.

[Tis09] A. Tisserand. "Function approximation based on estimated arithmetic operators." In *Signals, Systems and Computers, 2009 Conference Record of the Forty-Third Asilomar Conference on*, pp. 1798–1802, Nov 2009.

[WEL13] Lucas Wanner, Salma Elmalaki, Liangzhen Lai, Puneet Gupta, and Mani Srivastava. “VarEMU: An Emulation Testbed for Variability-aware Software.” In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS ’13, pp. 27:1–27:10, Piscataway, NJ, USA, 2013. IEEE Press.

[Yak76] Yoram Yakimovsky. “Boundary and Object Detection in Real World Images.” *J. ACM*, **23**(4):599–618, October 1976.

[Yet93] T. Yetiser. “Polymorphic viruses: Implementation, detection, and protection.” Technical report, VDS Advanced Research Group, 1993.

[YH94] Guangzheng Yang and Thomas S Huang. “Human face detection in a complex background.” *Pattern recognition*, **27**(1):53–63, 1994.

[ZGF11] Xia Zhao, Yao Guo, Qing Feng, and Xiangqun 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, pp. 641–646, New York, NY, USA, 2011. ACM.