NATIONAL UNIVERSITY OF SINGAPORE

DOCTORAL THESIS

Static Analysis Driven Testing of Performance and Energy-consumption Properties of Software

Submitted by:
ABHIJEET BANERJEE

Supervisor:
Professor Abhik Roychoudhury

Department of Computer Science
School of Computing
National University of Singapore

March 2016

Static Analysis Driven Testing of Performance and Energy-consumption Properties of Software

Abhijeet Banerjee
B.E.(Hons), IIEST, Shibpur

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY,
DEPARTMENT OF COMPUTER SCIENCE, SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE

2016

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis. This thesis has also not been submitted for any degree in any university previously.

Abhijeet Banerjee
23rd Mar 2016

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to my supervisor Prof. Abhik Roychoudhury for his continuous guidance and support throughout my graduate studies. I respect the commitment he has towards all his students. His timely feedback and continuous support were instrumental in advancing my research, as well as in writing this thesis.

I would like to thank Prof. Wong Weng Fai and Prof. Joxan Jaffar for their invaluable comments during my thesis proposal. Their encouragement has motivated to me further develop my research into meaningful solutions for real-life problems.

I would like to thank all my colleagues from System and Network Research Lab 1, System and Network Research Lab 2 and the TSUNAMI Lab. It has been a pleasure working with Dr. Sudipta Chattopadhyay, Lee Kee Chong, Dr. Clément Ballabriga and Dr. Hai-Feng Guo. In particular, I am very glad to have known Dr. Sudipta Chattopadhyay, who has been a colleague, a mentor and a good friend to me.

The journey from the qualifying examinations (QEs) to thesis submission can be a long one. But I am glad that I had the company of many good friends along the way. In particular, I am glad to have known Dr. Bablu Mukherjee, Dr. Marcel Böhme, Dr. Dawei Qi, Dr. Konstantin Rubinov and Dr Jooyong Yi. I will always cherish the deep discussions we had on all the topics scientific and otherwise. As for my soon to be Dr. friends, Sumanaruban Rajadurai, Nimantha Thushan Baranasuriya, Girisha Durrel De Silva and Lahiru Thilina Samarakoon, thanks for being there and making this long journey a memorable journey.

I wish to express my deepest gratitude towards my family for supporting me throughout my studies. Any of this would not have been possible without their support.

I would to like thank the National University of Singapore for giving me the opportunity to come and study at this wonderful institution. The facilities that we were provided for everything from education to sports and recreation were marvellous. In particular, the facilities at Central Library are exceptional. I have spent many happy hours browsing and reading the vast array of books there. I would like to extend my thanks to the Central Library staff for their dedication and effort.

Finally, I would like to thank A*Star and MoE for the "Scalable Timing Analysis Methods for Embedded Software" grant (Project Number 1121202007) and "Energy-aware Programming" grant (MOE2013-T2-1-115) , respectively. These grants where instrumental in conducting my research.

CONTENTS

ABSTRACT

Software testing is the process of evaluating the properties of a software. Properties of a software can be divided into two categories: functional properties and non-functional properties. Properties that influence the input-output behaviour of the software can be categorized as functional properties. On the other hand, properties that do not influence the input-output behaviour of the software directly can be categorized as non-functional properties. In context of real-time system software, testing functional as well as non functional properties is equally important. Over the years considerable amount of research effort has been dedicated in developing tools and techniques that systematically test various functional properties of a software. However, the same cannot be said about testing non-functional properties. Systematic testing of non-functional properties is often much more challenging than testing functional properties. This is because non-functional properties not only depends on the inputs to the program but also on the underlying hardware. Additionally, unlike the functional properties, non-functional properties are seldom annotated in the software itself. Such challenges provide the objectives for this work. *The primary objective of this work is to explore and address the major challenges in testing non-functional properties of a software.* To attain this objective, we have designed a technique that can be summarized into four key steps (i) identifying scenarios for suboptimal non-functional behaviour (ii) static analysis to identify potential program points that may lead to such suboptimal non-functional behaviour (iii) representing sub-optimal non-functional behaviour by means of assertions, at appropriate program points and finally, (iv) dynamic exploration of these assertions, guided by a well-defined coverage metric, in order to generate sub-optimal non-functional behaviour revealing test-cases. It is worthwhile to note that in our technique generation of assertions (in step three) is done in an automated fashion. In this work, we have presented instantiations of our technique for specific applications such as performance-stressing test-input generation for caches and energy-inefficiency revealing test-input generation for mobile apps. We also present a couple of follow-up works on energy-aware code re-factoring and energy-aware debugging to extend the support for energy-aware programming for mobile apps.

RELATED PUBLICATIONS

Abhijeet Banerjee. Static analysis driven performance and energy testing. In proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, Hong Kong, China, November, 2014

Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, Hong Kong, China, November, 2014

Abhijeet Banerjee, Sudipta Chattopadhyay and Abhik Roychoudhury. Static Analysis Driven Cache Performance Testing. In proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS 2013, Vancouver, BC, Canada, December, 2013

Abhijeet Banerjee, Sudipta Chattopadhyay, Abhik Roychoudhury. On Testing Embedded Software, In Advances in Computers, Elsevier, 2016, Volume 101, Pages 121-153, ISSN 0065-2458, ISBN 9780128051580

Abhijeet Banerjee, Hai-feng Guo and Abhik Roychoudhury. Debugging Energy-efficiency Related Field-failures in Mobile-apps In proceeding of the IEEE/ACM 3rd International Conference on Mobile Software Engineering and Systems, MOBILESoft 2016, Austin, Texas, USA, May 2016

Abhijeet Banerjee and Abhik Roychoudhury. Automated Re-factoring of Android Apps to Enhance Energy-efficiency In proceeding of the IEEE/ACM 3rd International Conference on Mobile Software Engineering and Systems, MOBILESoft 2016, Austin, Texas, USA, May 2016

Abhijeet Banerjee, Sudipta Chattopadhyay and Abhik Roychoudhury. Precise micro-architectural modeling for WCET analysis via AI+SAT. In 19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April, 2013

Abhijeet Banerjee and Abhik Roychoudhury. Energy-aware design patterns for mobile application development (Invited Talk), In proceedings of the 2nd International Workshop on Software Development Lifecycle for Mobile, DeMobile 2014, Hong Kong, China, November, 2014

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

Embedded systems are ubiquitous in the modern world. Such systems can be found in wide-variety of applications, ranging from mission-critical applications (such as pacemakers, Anti-lock Braking Systems), to casual applications (such as MP3 players and smartphones). Depending on the application domain, such systems may have to operate under one or more types of non-functional constraints. Two of the commonly observed non-functional constraints in such systems are performance constraints (due to the real-time nature of such systems) and energy constraints (due to the limited on-board battery capacity). Testing and validation of non-functional properties is an important aspect of quality assurance for such systems. However, until recently not many systematic techniques existed for automated testing of non-functional properties. Our work is an effort to address this need. In this chapter, we first introduce the reader to the key challenges in testing non-functional properties of a program. Subsequently, we shall present an overview of our framework that can be used for automated testing of non-functional properties in a program.

Software testing is an important part of the software development life-cycle. It is by conducting rigorous testing, one can be assured that the developed software meets the requirements and specifications of its user. However, due to the ever-increasing complexity of software systems, it is increasingly becoming impractical to adequately test software systems manually. The only practical way to alleviate this challenge is to devise tools and techniques that can automate software testing. Over the years, software engineering researchers have proposed tools and techniques that address different challenges associated with automating software testing. Our work is an effort to address one particular aspect of this challenge *i.e. automating software testing for non-functional properties*.

Software properties can broadly be divided into two categories: *functional* and *non-functional properties*. Properties that directly influence the input-output behaviour of the software are referred to as functional properties, whereas properties that do not influence the input-output relationship of the software are referred to as non-functional properties. Suitable example of non-functional properties are performance or energy-consumption of the system. In general, testing and validation of non-functional properties is very crucial for embedded systems. This is because such systems are usually resource-constrained (*e.g.* limited battery life such as in smartphones) and often have real-time constraints (*e.g.* timing deadlines such as in Anti-lock Braking System). Therefore, such systems must be tested for functional as well as non-functional properties. Over the years, researchers have developed a number of tools and techniques for *systematically testing* the functional properties of software, however the same cannot be said about non-functional properties. Systematic testing of non-functional properties is often much more challenging than testing functional properties. The following section describes some of the key challenges in testing and validation of non-functional properties.

1.1 CHALLENGES IN TESTING NON-FUNCTIONAL PROPERTIES

Testing non-functional properties is often more complicated than testing functional properties. This is primarily due to the fact that non-functional properties are not only influenced by the inputs to the program but also by the underlying hardware. As a result naive test-generation strategies, such as exhaustive testing, may be insufficient to provide appropriate information for the given non-functional property. Consider for example the scenario shown in Figure 1. This figures shows a simple program, with a single if condition (at line 3) within a *while* loop (at line 2). The *true* branch (at line 3) and the $false$ branch (at line 5) are taken in alternate iterations. Assume that memory block m_1 is accessed whenever the *true* branch is executed, whereas memory block m_2 is accessed on the $false$ branch. Also assume that both memory blocks m_1 and m_2 map to the same cache set (say cache set 1). In such a scenario, if the program is executed on a system with a direct mapped cache, memory blocks m_1 and m_2 will evict each other in alternate iterations, *i.e.* they would participate in cache thrashing. Such cache thrashing may significantly increase the execution time of the program. It is worthwhile to note that had the same program been executed on a system with 2-way, set associative cache there would have been no cache-thrashing. This example goes on to show that when testing non-functional properties it is important to account for the underlying hardware, on which the software would be executed. Modern embedded systems are equipped with a wide-range of hardware components. Research works that are targeted at the analysis and modeling of such hardware components (for the purposes of analysing non-functional properties), are commonly referred to as the micro-architecture analysis and are further described in this work.

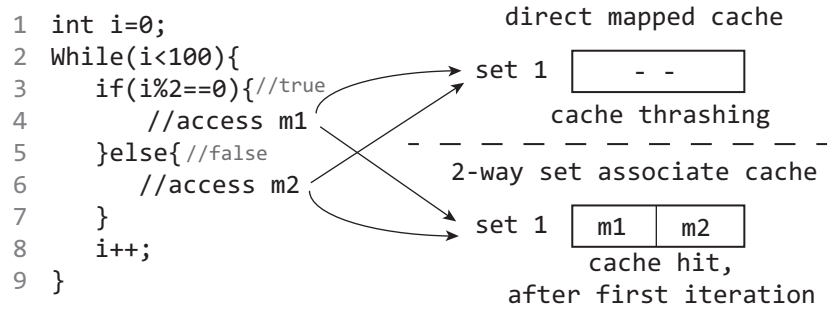


Figure 1: Estimating execution time in presence of micro-architectural components

Another challenge while testing non-functional properties arises due to the absence of appropriate coverage metric for a given non-functional property. Typically systematic testing frameworks would require a coverage metric to determine the completeness of the test-suite (with respect to the given property). However, due to absence of explicit annotation of non-functional properties in the program source-code, crafting such a coverage metric is often non-trivial.

1.2 ARE TESTING AND PROFILING THE SAME THING?

Often a common source of confusion is the interchangeable use of the terms testing and profiling. Profiling, in general, is the process of generating and recording the runtime program behaviour (for a given property of interest), for a given test-suite. The test-suite used to generate the runtime program behaviour, or the profile, is assumed to be representative of the entire input space of the program. However, manually generating such representative test-suite, even to uncover functional defects in a real-life program is often non-trivial. In the case of non-functional properties manually generating such a test-suite is almost impractical for most real-life program. This is because, as explained in the previous section, non-functional properties of a program depend on the inputs to the program as well as on hardware configurations of the system on which the program is executed. Systematic testing, on the other hand, can be used to explore the entire inputs space of the program and to generate test-inputs that highlight the property of interest (*i.e.* when the property of interest has been defined in an appropriate manner). Unlike profiling techniques, systematic testing techniques require a model of the system (software + hardware), at the required level of abstraction (*e.g.* control dependence graph, event flow graph). Figures 2 further illustrates the key difference between the two techniques. *In short, testing as in test-generation is not the same as profiling.*

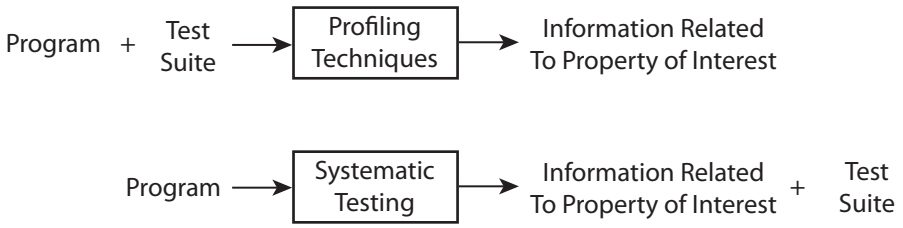


Figure 2: Key differences between profiling and systematic testing techniques

1.3 STATIC ANALYSIS DRIVEN TESTING OF PERFORMANCE AND ENERGY-CONSUMPTION PROPERTIES OF SOFTWARE: AN OVERVIEW

Existing software testing techniques can broadly be classified into two categories: techniques based on static analysis and techniques based on dynamic analysis. Techniques in both these categories have advantages and disadvantages of their own. For instance, techniques based on static analysis rely on various kinds of abstraction mechanisms so as to reduce the search space of the program, as a result of which such techniques tend to be scalable. This scalability, however, does not come free of cost. Often, static analysis based methods produce sound but imprecise results. Dynamic analysis techniques, on the other hand, can be much more precise but these methods often suffer from the problem of state space explosion (when the search space of the program is very large/infinite and the techniques takes impractical amount of time to explore it). Our test-generation framework uses both the static as well as dynamic analysis techniques at different stages of the analysis. This gives it the benefit of scalability as well as precision. The non-functional properties for which we instantiate our framework are *performance* and *energy efficiency*. In the following, we shall describe our framework around the following three topics:

i. Representing non-functional properties in a manner such that existing techniques from functional testing domain can be adapted for non-functional testing
ii. Identifying appropriate metrics to assist in exploration of non-functional properties
iii. Developing techniques for automated detection of inefficiencies related to non-functional properties

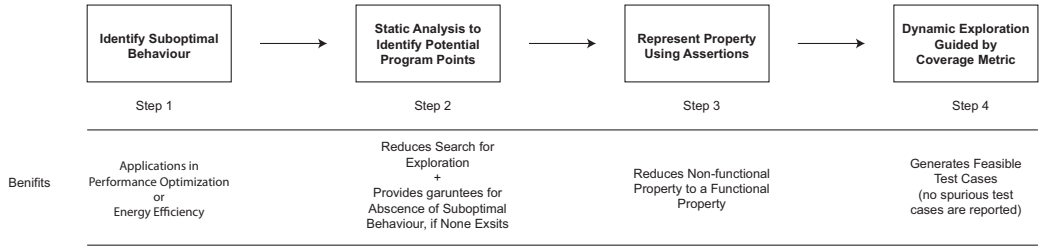


Figure 3: Key steps in our test generation technique

Our objective is to develop a technique that can be used for testing non-functional properties, specifically performance and energy-consumption. To obtain this objective we design a technique that can be divided into four key steps (Figure 3) (i) Identifying scenario for suboptimal non-functional behaviour (ii) Static analysis to identify potential program points that may lead to suboptimal non-functional behaviour (iii) Representation of non-functional properties as assertions, at appropriate program points and (iv) Dynamic exploration of assertions guided by a well-defined coverage metric. We discuss these steps in the following subsections.

1.3.1 Suboptimal Behaviour Identification

As with the development of any test-generation technique, we must first identify what constitutes as a suboptimal (or undesirable) behaviour. In particular, we shall identify suboptimal behaviour with respect to performance and energy consumption.

Performance (and execution time) of a program is dependent on the inputs to the program as well as on the states of underlying micro-architectural components(such as caches, pipelines, etc). Therefore, suboptimal performance of a program can be attributed to suboptimal performance of one or more of the underlying micro-architectural components. In one of our previous studies [1], we choose to focus on suboptimal performance due to caches. In particular, we focus on the scenario of cache thrashing. Cache thrashing can be described as a scenario when a frequently used cache line (or memory block) is replaced by other frequently used cache lines thereby causing a large number of cache misses (as a result suboptimal performance).

Identifying factors for energy-inefficiency in smartphone applications is important because such applications usually run on mobile devices that have limited amount of battery power. Additionally, such devices are equipped with a wide range of auxiliary hardware components, many of which may have an energy consumption higher than that of the CPU itself. Therefore, it is important to develop energy-aware programming and testing techniques for smartphone applications. However, until recently smartphone application development has been performed in an energy-oblivious fashion. Primarily because the major reasons for energy-inefficiencies in smartphone applications were not well understood. Therefore, in one of our works [2] we study (and categorise) the main reason for energy-inefficiencies in smartphone applications. Subsequently, we shall use the results of this study to identify scenarios of suboptimal energy

behaviour in smartphone applications. Existing studies, such as [3], have pointed out that I/O components (such sensor, GPS, Wifi, etc) play a substantial role in power consumption in smartphone applications. Another factor that affects the energy efficiency of smartphone applications is the misuse of power management utilities (such as Wakelocks in Android). Since I/O components as well as power management utilities can only be accessed through a set of API calls provided by the operating system, therefore presence of such API calls in an application could be an appropriate indicator for high energy consumption. However, high energy consumption does not necessarily imply the presence of energy inefficiency. Consider a scenario where the energy consumption is high due to high computation demand. Therefore, *to detect energy-inefficiencies one must look for scenarios where energy consumption is high but utilization (of device's components) is low*. Based on this intuition, we devised a *API-call coverage* guided test generation framework to explore energy-inefficiencies in Android applications [2]. The framework automatically explores a given application while simultaneously analysing the energy consumption to utilization ratio of the device. Based on the experiments conducted with our framework, we classified the prime reasons for energy inefficiencies into two categories: energy inefficiencies due to energy hotspots and energy inefficiencies due to energy bugs. An energy hotspot can be described as a scenario where executing an application causes the device to consume abnormally high amount of battery power even though the utilization of its hardware resources is low. In contrast, an energy bug can be described as a scenario where a malfunctioning application prevents the smartphone from becoming idle even after it has completed execution and there is no user activity. As a result of which the ratio of energy consumption vs utilization stays high, long after the user has navigated away from the application. Energy bugs are much more serious inefficiencies that energy hotspots because they cause a sustained energy loss from the device.

1.3.2 Static Analysis

Once we have identified the scenarios for suboptimal behaviour, we wish to generate test inputs that leads to such scenarios. Since non-functional behaviour depends on the program inputs as well as the underlying hardware states, the search space that needs to be explored to generate such test-inputs may be huge. Therefore, exhaustive exploration may often be impractical for such purposes. To overcome this challenge we first *statically* analyse the program using techniques based on the theory of abstract interpretation [4]. Such (abstract interpretation based) techniques analyse the abstract semantics of the program to estimate the property of interest. For instance, one example of property of interest could be presence (or absence) of a memory block in the cache, at a given program point. Abstract Interpretation based techniques are often very *scalable* because they analyse the abstract semantics of the program instead of its concrete semantics Also due to fact that the abstract semantics is superset of all possible concrete semantics of the program, therefore the results obtained are always *sound*. However, due to the use of abstraction the results obtained from such methods may be *imprecise* (overestimated). In our approach, we devise an abstract interpretation based technique to find out the potential program points that may lead to cache thrashing (when testing for performance) and energy bugs (when testing for energy consumption).

1.3.3 Representing Non-functional Properties

After we have obtained the potential program points that may have suboptimal behaviour we systematically generate assertions at all such locations. Each assertion is crafted such that its violation captures a scenario of suboptimal non-functional behaviour. For instance, when testing for suboptimal cache performance the violations of assertion captures a unique

cache thrashing scenarios. Similarly, when testing of energy inefficiency, violation of an assertion indicates the presence of an energy bug. Note that these assertions can be generated automatically from the results of the previous (static analysis based) step. It is worthwhile to know that by representing the non-functional properties, (such as presence of cache thrashing or energy bugs) as assertions, we *reduce the problem of non-functional testing to an equivalent functionality testing problem*.

One of the most important part of our technique is the formulation of the assertion. The exact formulation of the assertions depends on the non-functional behaviour being tested as well as the underlying hardware. For instance, when formulating the assertions for cache thrashing one has to account for the cache associativity as well as the cache replacement policy. Cache associativity can be used to estimate the number of memory blocks conflicting in the cache and cache replacement policy is necessary to find out the exact order in which the memory blocks would be evicted from the cache. In essence, all information that can influence the non-functional behaviour of the hardware component (in this example cache) must be known a priori.

1.3.4 Test-generation Through Dynamic Exploration

Existing dynamic exploration techniques, such as Directed Automated Random Testing (DART) [5] can be used to explore (and test functional properties of) a program without the need for writing specific test-cases. DART uses a combination of concrete and symbolic program executions to generate path-constraints for a given program execution. The constraints thus generated are systematically modified and solved to generate test-inputs that can be used to direct the execution along some previously unexplored path in the program. It is worthwhile to know that exploration techniques such as DART in their original form are only suited for checking the validity of functional properties. This is because the program source-code (or binary) alone may be insufficient for testing non-functional behaviour, such as performance. However, this is no longer a problem because with the addition of assertions (in the previous step) we have augmented the functional properties with the set of assertions capturing the non-functional properties as well. The instrumentation step therefore plays a crucial role in our test-generation framework. There is however another issue that needs to be addressed before we can start using functional testing technique such as DART for non-functional testing. It is worthwhile to know that a DART like exploration strategy starts from a random path in a program and keep exploring new paths until all feasible program paths have been explored. The exploration strategy in a DART like approach is completely oblivious to the presence of assertions (instrumented by us) in the program (i.e it does not take into account the presence or absence of assertions while making the exploration choices). Since we are primarily interested in checking the validity of assertions, hence such assertion-oblivious exploration strategy would be suboptimal for our purpose. Therefore, for our technique we devise an assertion-aware exploration strategy. Our technique computes a metric called *assertion-coverage*, that indicates the likely hood of finding unchecked assertions on a given program path. The algorithm then guides the exploration process towards a path that maximizes *assertion-coverage*. The intuition behind such an strategy is simple. Exploring paths that increase the net *assertion-coverage*, leads to maximum number of assertions being checked and therefore provides a greater likely hood of uncovering scenarios that lead to suboptimal behaviour. As a result of the assertion-aware exploration strategy our technique can explore maximum number of unique assertions within a given amount of time. Every time an assertions is encountered, its validity is checked. If an assertion is violated during the exploration, a suboptimal performance/energy consumption issue is recorded along with a symbolic formula capturing the set of inputs that leads to the violation of that assertion.

It is worthwhile to know that the unlike the static analysis phase, the dynamic exploration phase of our framework is path-sensitive, due to which all test cases generated by the dynamic exploration phase are *real* scenarios of suboptimal non-functional behaviour. The test cases generated by our framework can be used to optimize non-functional behaviour of a program. More specifically, for improving performance, the results from our framework can be used for design space exploration and for developing input-sensitive cache locking techniques which can provide better performance gains as compared to traditional cache locking techniques such as [6]. For improving energy efficiency, the results from our framework can be used for developing automated techniques for energy-efficient repair code generation .

1.4 KEY CONTRIBUTIONS

The primary objective of this work is to explore and address the major challenges in testing non-functional properties of a software. In particular, we focus on the non-functional properties of performance and energy-consumption. Systematic testing of non-functional properties is often much more challenging than testing functional properties because non-functional properties not only depends on the inputs to the program but also on the underlying hardware. Additionally, unlike the functional properties, non-functional properties are seldom annotated in the software itself. Such challenges provide the objectives for this work. The key contributions of this work can be summarized as follows:

1. Performance
    a. We propose a test-generation framework that exposes the cache performance issues of an embedded software to the developer. One appealing nature of test suite generated by this framework is that it does not include any spurious test cases (i.e. a test-case that does not capture a cache performance issue in any feasible execution). Our test-generation framework is guided by a well-defined coverage metric that assists in uncovering cache-thrashing scenarios in a systematic fashion.
    b. We demonstrate the use of an assertion-based approach by which non-functional behaviour (such as cache thrashing), that is not explicitly encoded in the program source code, can be represented as functional properties and thereby enabling the use of variety of functionality testing tools for the purposes of non-functional testing as well. It is worthwhile to know that in our test-generation technique the assertions, that are used to represent the non-functional property of interest, are generated and instrumented into the program in an automated fashion.
    c. We further show the utility of our performance-stressing test-generation framework in applications such as design space exploration and performance optimization.

2. Energy-consumption
    a. We present one of the first systematic definition for energy-inefficient behaviour in mobile apps. We also introduce a new metric of *E/U ratio* ($\frac{EnergyConsumption}{Utilization}$) that can be used to measure energy-inefficiency of a given app. The intuition behind the metric of *E/U ratio* is that higher utilization (of system resources) for a given energy consumption is more energy-efficient than lower utilization (of system resources) for the same amount of energy consumption. In essence, a higher *E/U ratio* signifies more inefficiency.

b. We introduce a fault-model for energy-inefficient behaviour in mobile apps. In particular, we categorize energy-inefficiency in mobile apps into two categories: *energy hotspots* and *energy bugs*. An anomalously high *E/U ratio* exhibited by the mobile device during the execution of an app indicates presence of an *energy hotspot* whereas the scenario where the mobile device exhibits a high *E/U ratio* even after an app has completed execution, indicates presence of an *energy bug*. We also provide real-life examples for each type of *energy hotspot* and *energy bug* in our work.

c. Based on our understanding of different types of energy-inefficiencies in mobile apps, we were able to develop various tools and techniques that can assist a programmer to do energy-aware programming. In particular, we provide tools and techniques for systematic energy-aware testing, energy-aware re-factoring and energy-aware debugging (of field-failures), in mobile apps.

1.5 ORGANIZATION OF CHAPTERS

This work targets at different aspects of non-functional testing. Depending on the nature and application of an embedded system, different non-functional properties may be of interest. However in this work, we specifically focus on two non-functional properties: performance (crucial for real-time systems) and energy-consumption (important for battery-constrained, mobile devices). We start by describing some of the key concepts and existing works on performance analysis in Chapter 2. We also describe some our efforts to improve the state-of-art in performance analysis in Chapter 2. Subsequently, Chapter 3 presents one of our works that uses a combination of static and dynamic analysis to automatically generate test cases that lead to inferior cache performance. It is worthwhile to know that this was one of the first works to propose a systematic technique for non-functional test generation. Chapter 4 introduces the reader to basic concepts and existing research work on the topic of energy-consumption analysis. Chapter 5 presents a grey-box testing approach for automatically exploring and detecting energy-inefficiencies in mobile apps. More importantly the work presented in Chapter 5 describes what it means to exhibit *bad* energy consumption behaviour, for mobile apps. This understanding is further used to define a fault-model for energy-inefficiency is mobile apps. This fault-model provides the ground work for the framework presented in Chapter 6, which presents a white-box testing approach to automatically detect, validate and repair energy bugs in mobile apps. Chapter 6 also introduces the tool EnergyPatch which can be used by app developers to test and repair their apps before deployment. We also present a couple of follow-up works in Chapters 7 and 8 on energy-aware re-factoring and energy-aware debugging in the context of mobile apps. Finally, we conclude this thesis in Chapter 9 with a brief discussion on the contributions of this thesis and a potential future work direction.

2 PERFORMANCE ANALYSIS: BACKGROUND & LITERATURE REVIEW

This chapter introduces the reader to some of the key concepts in performance analysis. It briefly describes the various commercial tools (such as ARM Streamline Performance Analyzer, Intel VTune Amplifier, *etc*), available for the purposes of performance analysis. It is also describes some of the existing research works related to performance analysis. In particular, the works on performance analysis are described in three different parts (i) works on performance profiling (ii) works on performance estimation and finally (iii) works on performance testing. We also present a new micro-architectural modeling framework that uses abstract interpretation and satisfiability checking to generate worst-case execution time (WCET) estimates for a given program. This framework can be used to substantially improve the accuracy of WCET analysis in the presence of many infeasible paths in the program.

2.1 REAL-TIME EMBEDDED SYSTEMS

Embedded systems represent the class of computer systems that are designed for a specific application. Often such application involves the controller (or the computer) controlling a custom piece of hardware (usually an electro-mechanical component). Embedded systems come in variety of designs and complexities. They can be simple systems used for controlling common, household appliances such as washing machines and dishwashers or they can be complex, mission-critical medical equipment such as a pacemaker. Depending on the application domain of such systems, they may have performance or timing constraints. These timing constraints are often real-time in nature, hence referred to as real-time constraints. Having a real-time constraints means that such systems should not only be capable of processing the correct output for a given input, but it should do so within a given deadline. Inability to complete a task within the deadline may cause a degradation in the Quality-of-Service or even catastrophic consequences, in certain applications. Depending on the seriousness of the real-time constraint, such systems are further classified into two categories: *hard real-time systems* and *soft real-time systems*.

Hard real-time systems are computer systems which can not afford to miss even a single timing deadline. Missing a deadlines in such systems can lead to catastrophic consequences. For example, if a pacemaker fails to provide the right amount of electrical impulse, at the right time, the patient's heart may stop functioning, leading to fatal consequences.

Soft-real time systems in contrast, can afford to miss a few timing deadlines and may still keep functioning. However, meeting all timing deadlines is highly desirable as missing a deadline may lead to degraded Quality-of-Service. An example of such system would be a live video playback system (video encoder/decoder). Such systems are used widely for showing live feed from sports events. It is highly desirable, to have an un-interrupt video stream of the sports event but occasionally missing a few video frames should not cause any catastrophic consequences.

2.2 OVERVIEW OF PERFORMANCE ANALYSIS TOOLS

Computer programmers and designers often use a number of profiling tools to understand and optimize the system behaviour. Many commercially available profilers present today, use sampling or instrumentation based techniques to profile performance or power consumption of a system. In this section, we briefly describe some of the commercially available profilers, which include *ARM Streamline Performance Analyzer*, *ARM µVision4 IDE*, *Intel VTune Amplifier*, *AMD CodeAnalyst*, and *Valgrind*. The above mentioned profilers are targeted at different platforms and provide wide range of features, however the common feature amongst these tools is that they all perform dynamic program analysis. Recall that dynamic program analysis, unlike static program analysis, comprises of executing programs on the target system.

Streamline Performance Analyzer, is a part of the *ARM DS-5 toolchain*. This is a GUI-based tool and it is primarily targeted at the *Cortex-A* series and *Cortex-R* series of *ARM* processors. It uses various performance counters and sampling-based techniques to capture profiling data. The tool supports two modes of sampling, which are timer-based sampling and event-based sampling. Quantitative properties such as cache-misses or cache-hits can be statistically assigned to particular process or a thread in a program. Additionally, the streamline analyzer, together with the *ARM energy probe*, can be used to capture power consumption of a program. The *ARM energy probe* is essentially a USB device which can sample voltage, current and power. The sampled data is then synchronized with the software execution trace and various performance metrics, to give the developer an idea about the energy hotspots in the program. This tool also has features to support profiling on symmetric multiprocessor (SMP) platforms. It also displays the observed thread activity on a specific cores. This gives the programmers an intuition of how their code is distributed across different cores.

µVision4 IDE is another GUI-based, embedded applications tool marketed by ARM Holdings plc. This tool is targeted at the *Cortex-M*, *Cortex-R4*, *ARM7* and *ARM9* processor-based devices. The performance analyzer in the *µVision4 IDE* is capable of recording the time spent for executing a particular function in a program. Additionally, this tool can generate the execution trace related information, for a given program. However, unlike the *Streamline Performance Analyzer*, *µVision4 IDE* has no support for multi-core processors or power consumption profiling.

VTune Amplifier, is a performance profiler developed and marketed by Intel Corporation. This tool is primarily targeted at the systems with Intel processors, although some of the basic features of this tool can be used for profiling other systems as well. The tool used event-based sampling, performance counters and call-graph profiling to generate the profiling data. The profiling results can be visualized through a GUI, on a per-process level, per-thread level or a per-module level, with the resolution of a single instruction. For power analysis, the tool has two built in modes, CPU sleep state and CPU frequency scaling. The amount of time spent by the CPU in sleep-state and the frequency at which the CPU operates are key factors (but not the only factors) for determining average power consumption. Therefore, profiling data generated from these two modes can give a rough-estimate of the average power consumption, while executing a program (or a module).

AMD CodeAnalyst, is a GUI-based performance profiler targeted at *x86* and *x86-64* based systems. Additionally, the features provided by this tool support multi-core and non-uniform memory access (NUMA) systems. *CodeAnalyst* is based on statistical profiling tool *OProfile*. It uses various hardware based profiling techniques as well some generic timer-based profiling

techniques, to capture the profiling data. However, some of the hardware based profiling techniques are specific to AMD processors and therefore applicable to systems with AMD processors only. The profiling data can be examined on a per-function level or per-thread-level, with an instruction-level resolution.

Valgrind Tool Suite is a set of debugging and profiling tools, targeted at a large number of processors (such as *x86*, *x86-64* and *PowerPC*, *ARMv7*) and is avialable under the GPL license. The tools in *Valgrind* generates debugging/profiling related information using runtime instrumentation. In particular, the *Cachegrind* and the *Callgrind* tools from the *Valgrind* Tool Suite, are useful for performance analysis. *Cachegrind* is essentially a cache profiler, which performs detailed simulation of caches to highlight source of cache misses in the program. It is capable of generating summary for the memory references, cache misses and the instructions executed for each line of source code. The results can be examined at per-function level, per-module level or for the entire program. *Callgrind* is an extension to *Cachegrind* tool and it provides additional information related to call-graphs. *KCachegrind*, is the GUI-based version of the tool *Callgrind* and it is also available under the GPL license.

2.3 APPROACHES USED FOR PERFORMANCE TESTING/ ESTIMATION

There can be a number of reasons to analyse the performance of a given system. For instance, in the case of hard-real time systems knowing the upper bound on execution time is very important. For such systems, techniques for worst case execution time (WCET) analysis can be very useful. For other systems, performance may directly correlate to the quality-of-service and hence the developer may want to fine-tune the system performance. In such scenarios, the developer can use a performance-aware test generation technique to find out the test inputs that degrade performance and subsequently, either change the program or the hardware configuration to optimize the system performance. It is also possible that the developer/tester wishes to compare the performance of a program on two different platforms (hardware configurations). For such scenarios, profiling techniques can be useful. Profiling techniques can be also used to approximate the performance trend for a given program. Table 1 shows some of the key aspects of the different performance analysis methodologies.

Table 1: Key aspects of different performance analysis methodologies

	Profiling	Test Generation	Estimation
Estimates upper, lower bound on execution time	No	No	Yes
Test inputs needed to conduct performance analysis	Yes	No	No
Find test inputs for suboptimal behaviour	No	Yes	No
Underlying framework	Dynamic Analysis	Dynamic + Static Analysis	Static Analysis

2.4 PERFORMANCE PROFILING TECHNIQUES

Profiling can be described as a dynamic analysis techniques where a program is executed for a set of representative inputs to observe the program behaviour. Such profiling techniques often work on full or compressed execution traces to extract useful information about the program behaviour. It is assumed that the representative inputs for obtaining the execution traces are known beforehand. Many commercially available profilers (such as ARM Streamline Performance Analyzer, Intel VTune Amplifier and AMD CodeAnalyst), use sampling or instrumentation based techniques to profile performance or energy consumption of a program. However, for the purpose of brevity, in this section we shall restrict our discussion only to existing research techniques based on profiling.

Profiling based techniques, in general, do not perform micro-architectural modeling (systematic analysis of underlying hardware components). As a result they are less complex and light-weight. However, since they do not model (or take into account) the underlying hardware, they cannot provide any guarantees on the upper or lower bounds on the execution time of a program. Hence they are not very useful for analysis of hard real-time systems (which require strict timing guarantees). Another challenge while using profiling based techniques is *completeness*. For instance completeness in terms of program-paths would mean that all program-paths have been executed at least once. However, in practice a complete path coverage is seldom achieved because the number of paths in a program increases exponentially with the number of decision variables. (A program with η decision variable can have 2^{η} paths.) Most profiling based techniques execute the (analysed) program only for a subset of program-paths. Despite these limitations, profiling based techniques have been extensively used for a number of practical purposes. In particular, systems (such as soft, real-time systems), which do not require strict guarantees on performance, can benefit from the use of profiling based techniques. In the following paragraphs, we shall see to two extensions of the profiling based works; extending profiling to estimate WCET and extending profiling techniques to estimate program cost.

As mentioned in previous paragraphs, inherent limitations of profiling make it almost impractical for bounding the execution times (*i.e.* WCET). However, a number of works [7],[8] have explored ways to overcome the aforementioned limitations. The work of [7] in particular measures the execution times of small program segments and subsequently stitches them together to estimate the overall execution time of the program. It is worthwhile to note that the technique of [7] does not model the underlying hardware while profiling. This raise some concerns over the correctness of generated results through this approach. To address these concerns, [7] tries to use compiler-level techniques to reduce variability throughout different executions. Another work [8], introduces a technique for WCET estimation for probabilistic, hard, real-time systems. [8] suggests that a hard, real-time system must meet the deadlines with a high probability. It proposes a framework, where execution profiles of smaller program units are probabilistically combined to estimate the worst case execution time of the entire program. Although, their method might generate better WCET estimates than the conventional end-to-end measurement based techniques, but it should not be used to determine the WCET of a hard real-time system, because by definition hard real-time systems *must* meet all the deadlines, under all conditions.

Recent advances in profiling [9, 10] have extended on the traditional profiling techniques to compute the performance behaviour of a program by means of an approximate cost function. The cost function relates program inputs with the overall cost of the program execution. It is worthwhile to know that such cost functions are approximations and do not necessarily capture the actual cost of executing the program for a given input.

2.5 PERFORMANCE ESTIMATION TECHNIQUES

Static analysis methods refers to the set of timing analysis techniques, which estimate a bounds on the execution time of the program, without actually executing it on the real hardware. Many quantitative properties of real-time software such as worst-case execution time (WCET) and best-case Execution Time (BCET) are often undecidable. But a sound knowledge of such properties for real-time software is essential. Therefore, static methods are used to generate sound but an over-approximated estimates for such properties of a program. The process of estimating static timing analysis can be divides into following three phases : Control Flow Analysis, Micro-architectural Analysis, Estimate Calculation (see Figure 4).

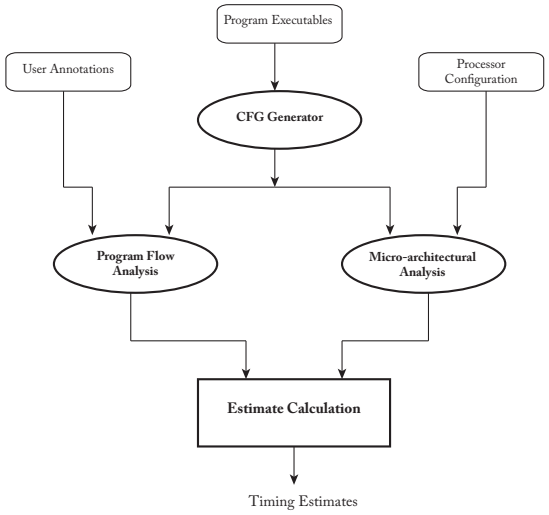


Figure 4: An abstract illustration of a timing analysis framework

2.5.1 Program Flow Analysis

Program Flow Analysis is a term often used to represent the set of techniques, which are used to derive constraints on the paths of a program's control flow graph (CFG). Flow analysis take in the program CFG as an input and analyses it to generate various flow related information, such as loop bounds and infeasible paths in the CFG. In general, it is difficult to precisely calculate the program flows statically. Therefore, a safe-over approximation of the flow related information is estimated. For example, in order to calculate the WCET of a program containing loop, an upper-bound on the number of loop iteration must be known.

Previous research work such as [11], [12] propose techniques for automatic loop bound detection. Infeasible path detection is also an important part of flow analysis. Unlike loop bound detection, infeasible path detection is not a necessity for WCET estimation. But it is highly desirable, because in the presence of infeasible paths, the WCET can be over-estimated. For instance, consider a program CFG which has an infeasible path Ip. Also suppose Ip has the longest execution time over the whole CFG, (say T_{Ip}). In absence of any infeasible path related flow information, the WCET analyser would report the WCET to be T_{Ip}. Clearly, since Ip is never executed in any concrete execution of the program, WCET of the program is over-estimated. Additionally, in the presence of infeasible paths, micro-architectural analysis can be imprecise, which can also lead to an over-estimated WCET. Methods such as the one proposed in [13] can be used for automatic infeasible path detection. Flow information such as loop bound and infeasible paths can also be added manually, as annotations or user-provided constraints.

2.5.2 Micro-architectural Analysis

Most modern processors uses a number of performance enhancing features such as caches, pipeline and branch predictors. These features are very useful for increasing performance, but they also make the task of timing analysis complicated. Worst case execution time (WCET) of a program is directly influenced by the various micro-architectural components. So, in order to produce a safe and precise estimate of a program's WCET, micro-architectural analysis must be performed. The following paragraphs describe some of the existing research work for analysing various micro-architectural components such as caches, pipelines and branch predictors.

Caches are fast memory used in modern computer systems, to hide the latency of slower memory access. The presence or absence of accessed memory blocks in the cache can influence the execution time of a program. But for most programs it is statically undecidable to accurately calculate the contents of the cache, at a given program point. Fortunately, static analysis based methods can be used to estimate an over-approximation of cache contents, at a given program point in the CFG.

The work in [14], proposes one of the first method to model the behaviour of instruction caches. In their method they construct a cache conflict graph to model the inter-instruction conflicts. This graph is used to generate constraints representing the cache behaviour. They also suggest a method to represent the structural and functional properties of the program as linear constraints. Structural constraints are derived from the flow analysis of the program CFG. For example, the information that the execution count of a basic block is equal to the number to time control flow edges enters the basic block can be represented by a structural constraint. On the other hand, information such as the upper bound on number of iterations of a loop are represented by functional constraints. Their approach provides an elegant way to represent all the flow related information as well as micro-architectural behaviour as a system of linear equations. An integer linear programming (ILP) solver can then be used to obtain the worst case execution cycles of the program. A major limitation with their approach is that as the associativity of the caches increase, the ILP problem increasingly gets more complex and therefore takes a long time to solve.

One of first scalable approaches for performing cache analysis was proposed in [15]. They presented an *abstraction interpretation* based approach to categorize the memory blocks in the cache. The abstract semantics used in their framework consisted of an abstract domain of caches states and a set of abstract functions. The abstract domain of cache states represents an approximation of the set of all concrete cache states at a given program point, whereas, the set of abstract function consists of an abstract *Join* function (used to merge multiple abstract cache states into a single abstract cache states) and a abstract *Update* function (reflects the side-effects of a cache reference on the abstract cache state). The authors use three different approaches to perform cache analysis, they are Must, May, Persistence analysis. Must analysis can be used to identify the *always-hit* (AH) memory block in the cache. Access to AH memory blocks always results in a cache hit. Likewise, May analysis can be used to identify *always-miss* memory blocks in the cache. Access to AM memory blocks always results in a cache miss. Persistence analysis can be used to identify memory blocks, access to which will always results in a cache-hit, except for the first access for which it would be a cache miss. To get a more precise cache classification, their cache analysis framework employs VIVU (virtual inlinning virtual unrolling). This is especially beneficial for program having loops and recursive procedure calls as it helps in isolating the first iteration of a loop (or recursive call) from the renaming iterations (or recursive calls). Due to the use of abstract interpretation for micro-architectural analysis, this approach is much simpler than the approach suggested by [16], especially for set-associative caches.

The work of [17], extends the framework proposed in [15], to multilevel, non-inclusive, set-associative caches. In order to model the cache behaviour for multi-level caches, a *cache access classification* (or CAC) has been suggested in this paper. A CAC is utilized to safely estimate whether an access to a memory block occurs at cache level L. The CAC classification for a memory block can be *Always Accessed*, *Never Accessed* or *Uncertain*. CAC for a cache level L depends directly on the CAC and the cache-hit-miss-classification (CHMC) classification of the cache level L-1. If a memory block is classified as AH in cache level L-1, that memory block would not be referenced in the cache level L and therefore it is classified as *Always Accessed*. Likewise, *Never Accessed* memory block can also be identified. All memory blocks which can not be safely classified as *Always Accessed* or *Never Accessed* have a CAC of *Uncertain*. All blocks access the cache at level 1, therefore all blocks have a CAC of *Always Accessed* CAC for cache level 1. CHMC for level 1 cache is calculated using the technique proposed in [15]. For other cache levels, after the CAC has been calculated, CHMC can be calculated in a similar manner.

Abstract interpretation based micro-architectural analysis method are fast and scalable, but the result of analysis (WCET estimates) can be imprecise. Since abstract interpretation is inherently path in-sensitive, it cannot distinguish between feasible and infeasible paths. In the presence of infeasible paths, abstract interpretation based cache analysis might lead to infeasible cache states. Although, the presence of such infeasible cache states does not affect the correctness of the analysis but the WCET of the program might be overestimated. The work of [18], proposes a method to improve the precision of abstract interpretation based instruction cache analysis, with the help of model checking. First the cache analysis proposed in [15] is used to obtain the CHMC for the memory blocks. After obtaining the CHMC for the memory blocks, repeated runs of model checking are applied to efficiently refine the WCET estimate. Given a potentially conflicting pair of blocks, a model checker (CBMC) is used to verify if the pair actually conflict in any execution. If the conflict was indeed spurious (possibly due to infeasible path in the program), the classification of the memory block is adjusted. An advantage of this approach is that the refinement of cache categorization can be stopped at any time and still the obtained WCET estimate would be sound.

Similar to instruction cache analysis, data cache analysis is also an important part of micro-architectural analysis. But the process of analysing data caches is much more challenging than analysing instruction caches. This is because, unlike instruction cache analysis, for any memory access, multiple memory blocks might be accessed at different instances of the access. Many of previous research work, overcome this challenge by using techniques such as *Address Analysis*. *Address Analysis* generates an over-approximation of the set of addresses that can be accessed by a specific load/store instruction, in a program.

The study by [19] extends the abstract interpretation based framework (by [15]), for data caches. In particular, they suggest an approach to apply *Persistence Analysis* for data caches, although neither details of address analysis nor any experimental results were presented in this paper. [20] presents a technique for address analysis and they use the results of address analysis to perform *Must analysis* on data caches. The original must analysis described in [15] can not be directly applied to data caches because the each program point might be associated with a range of memory blocks. They also unroll the loop partially to improve the precision of their framework, but this makes their analysis more expensive. [21] present an approach to apply *May analysis* for data caches. Their work also presents one of the first frameworks to model unified caches. Unified caches are used to store data as well as instructions. Unified caches have been used in commercial processors such as *Intel Itanium 2*, *Intel Core i7* and *AMD Phenom II*.

[22] extended the persistence analysis to multi-level, set-associative data caches. A Cache access classification (CAC) is used to find out the memory references for a particular cache

level. The CAC for multi-level, data cache analysis is similar to the CAC define for multi-level, instruction caches analysis proposed by [17] A recent work by [23], has shown that the persistence analysis for data caches as proposed by [19], is not safe for WCET estimation. The original persistence analysis has a flaw in the abstract *Update* function, due to which it could underestimate the WCET. This flaw can be corrected by keeping track of *Younger Set* for each memory block. A *Younger Set* for a memory block m, denotes the set of memory blocks which may be younger, than m along any execution path. Additionally, [23] proposes a scope-aware, persistence analysis for data caches. Scope-aware, persistence analysis exploits the temporal scopes of the memory blocks to reduce the pessimism in the data cache analysis. A temporal scope of a memory block m, captures the loop iterations of the program, where m can be accessed. In particular, if two memory blocks m_1 and m_2 map to the same cache set, but have different temporal scopes, they would not conflict in the cache. Essentially, the original persistence analysis for data cache estimates the global persistence of a memory block whereas the scope based persistence analysis only estimates the persistence of memory block within its temporal scope. Therefore, scope-aware persistence analysis can lead to more precise WCET estimate.

Pipeline Analysis : Advanced feature such as pipeline are a boost to performance but at same time it makes the process of WCET estimation very challenging. In pipelined processors the execution time of instructions might overlap, due which a execution time of a basic block cannot be obtained by simply adding up the execution time of constituent instructions of the basic block. In order to obtain a *safe* WCET estimate, the micro-architectural analysis must analyse the effects of pipeline along with other components such as caches and branch-predictors. Previous research work has presented techniques to model the pipeline behaviour, for various processor architectures. Some of those works would be discussed in the following paragraphs.

The work in [24], proposes a framework for WCET estimation in presence of pipelines and caches. The target processor for their study was MIPS R3000, which has a simple five-stage pipeline. The effect of pipeline is modeled by using a reservation tables of resources for each instruction. The reservation table is used to analyse timing interaction between instructions inside and across the basic block. A bottom-up algorithm is used to find worst case execution time estimates for a path. WCET for the whole program can be estimated using concatenation of paths. Every time a instruction is added to a path, a new reservation table is computed. Reservation tables are compacted whenever possible by keeping information only from the beginning and the end of a path. This framework of [24] is later extended in [25], for multiple-issue architectures.

The micro-architectural analysis proposed in [26] also focuses on pipelines and instruction caches. The target processor for their experiments was MicroSPARC 1, which has a very simple, in-order pipeline. The analysis proceeds by first determining the cache-hit-miss categorization (CHMC) using static, cache simulation. Each instruction is associated with the set of registers which it can access. Additional information, such as the maximum number of cycles per pipeline stage and the first and last pipeline stage, from which forwarding can take place is stored for each instruction. The earliest and latest usage time for register files is also tracked in order to avoid data hazards. Program path analysis is done by concatenation of instruction on a path. Loops are analysed using a bottom up approach, beginning with the inner-most loop All the paths through the loop are merged and this information is used to the analyse the outer loops in an iterative manner.

[27] present an integrated, integer linear programming (ILP) based approach for analysing the cache and pipeline behaviour. The processor model used for their experiments was Intel i960KB, which has a very simple pipeline. The simple architecture of the pipeline allowed

the authors to restrict the model to look for structural hazards only. Therefore, their approach would be much difficult to apply for superscalar, out-of-order processors.

The work of [28] presents an abstract interpretation based approach for modeling pipelines, for in-order, superscalar processors. This key challenges while modeling pipeline in superscalar processors is that in such architectures resource allocation to the instruction occurs dynamically. Due to this reason a static reservation table based approach as proposed in earlier works would be inadequate to model superscalar processor pipelines. The target processor for this study was Sun SuperSPARC 1. The modeling abstracts the pipeline by representing the set of all pipeline states, at a program point, by a single abstract pipeline state. Although the abstract semantics proposed in this paper was described only for in-order pipelines but they can be extended to out-of-order pipelines as well.

The work in [29], proposed an abstract interpretation based analysis, for an architecture with an in-order pipeline, caches and branch prediction. The target processor for their study was Motorola ColdFire 5307. Their approach decomposes the pipeline into several smaller and simpler units. These units (of the pipeline model), may represent single or a combination of actual pipeline state. Additionally, these units can communicate with each other and the memory by using signals. In their model, an abstract pipeline state represents a set of concrete pipeline states. The abstracted model of pipeline groups together those states which have similar timing behaviour. But such assumption has its own limitations, consider a scenario, where the execution time of an instruction is not known statically (some instructions execution time depends on the operands). In such a scenario, in their model, a pipeline state can have several successor pipeline states, to account for the variable execution time. This feature is the prime limitation of their model because it can cause state space explosion problem.

The work in [30], presents a ILP based approach to model an out-of-order, superscalar processors with pipelines. To estimate the WCET of basic block, first it is analysed in isolation (assuming an empty pipeline at the beginning of the execution). In absence of instructions from any other basic block, a basic block will take maximum time to execute when there is maximum resource contention amongst the instructions within the basic block. Based upon this observation, a rough estimate for earliest starting time and latest finish times for a instruction is estimated, in the presence of maximum possible delay. Note that some contention (and possible delays), can be ruled out due to data dependencies. As a result of ruling out some contentions, the earliest start times and latest finish times can be refined. The process of refinement continues, iteratively, until a fixed point has been achieved. This gives an estimate on WCET of a basic block when executing on an empty pipeline. But usually some instruction would be executing before and after a basic block (termed as prologue and epilogue of an basic block in the paper). The delay caused to due to possible contention between instructions in prologue and epilogue is estimated conservatively. The basic block estimates are then used in the ILP formulation to obtain the WCET of the whole program.

Branch Prediction Analysis : Another micro-architectural component which can affect the execution time of a program is a branch predictor. Most superscalar processors use some variation of branch prediction mechanism, to predict the result of a branch instruction. Such a component is required because superscalar microprocessor pre-fetch instruction to keep the pipeline busy. As long the instructions are in sequential order, pre-fetching is straightforward. But when a branch instruction is encountered pre-fetching (and hence the pipeline) is stalled, until the target address of the branch instruction is calculated in the later stages of the pipeline. In order to prevent the pipelines stall, a branch predictor is used to speculate the outcome of the branch instruction.

Due to the unpredictability introduced by branch prediction many processors targeted for embedded application, employ some trivial form of branch prediction or no branch prediction at all. [31] presents a brief overview of various branch predictions mechanisms for some of

the commonly used processors. Although this study does not present any new techniques to statically estimate the WCET of program in presence of branch prediction, but it does gives some idea about the impact of various branch prediction mechanisms on processor performance. [31] compares the branch prediction mechanism of Intel Pentium III, Intel Pentium 4 ,AMD Athlon, Sun UltraSparc II, Sun UltraSparc III, NEC V850 and ARM7. The NEC V850 and ARM7 microprocessor are primarily designed for use in embedded application and their branch prediction mechanism is very regular. As a result static analysis of programs, for these processor architecture is very straightforward. The cost of high predictability in such processors is paid by sacrificing performance. Since embedded application are increasingly becoming more performance intensive, processors with better performance and more aggressive branch prediction mechanisms must be used. Existing research work such as [32], [33], [34] suggest methods for timing analysis of programs on processor architectures with branch prediction. [32] suggest an abstract interpretation based micro-architectural framework to analyse set-associative, Branch Target Buffers (BTB), for Pentium architecture. [33] extends the ILP based framework proposed by [35], to integrate various global branch prediction schemes. They also suggest one of first approaches to model instruction caches in presence of speculative execution. Speculative execution can lead to a behaviour called *wrong-path cache effect*. This behaviour can be described as pre-fetching of cache instructions due branch mis-prediction. Although, this study proposes an elegant method for representing constraints generated by program flow analysis, cache modelling and branch-prediction modelling, but the framework has been applied only to direct mapped caches. Another work [34] proposes an abstract interpretation based framework to model branch target instruction caches (BTIC). They also present an instantiation of their framework for the Motorola PowerPC 56x which has a fully-associative, 8-way, tgt-BTIC. However, since the framework proposed in [34] is modular it might be possible to adapt the framework for analysing other kinds of branch target buffer as well.

The methods mentioned in the above paragraphs describe some of the exiting frameworks to model *dynamic* branch predictors. Another line of work exists, which presents an approach to model *static* branch predictors. Recall that *static* branch prediction mechanisms, have a regular prediction pattern, irrespective of the code (for example BTFN). This property can be exploited by compilers to the benefit of static analysis. One such approach is proposed by [36], which suggests the use of compiler optimizations, to increase the predictability of static branch prediction techniques. Their algorithm for WCET estimation is composed of two key steps. In the first step, all conditional branches which have not been statically predicted are assumed to be mis-predicted and the worst case execution path and the WCET is computed. In the second step, all conditional branches which are still unpredicted, are statically predicted. The algorithm iterates over these two steps until the WCET is stabilized.

2.5.3 Estimate Calculation

The output of the program flow analysis phases and the micro-architectural analysis phases are used as inputs to this phase. The purpose of this phase is to calculate the longest, feasible, path in the program CFG. As a result of the analysis performed in this phase, an upper bound on the execution time of a program can be estimated. Most of the existing methods for bound calculation can be classified into following three categories

- Path based methods
- Tree based methods
- Implicit Path Enumeration based methods

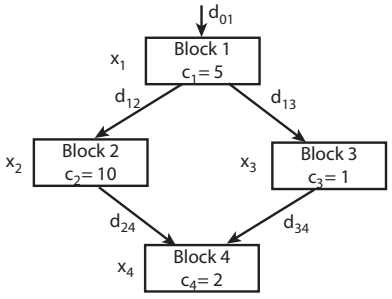
Path Based Methods primarily search for the program path with longest execution time, in the control flow graph of a program. Interestingly, the longest structural path in the program CFG does not always have the longest execution time. In the paths based methods, the task of finding the longest path is achieved by enumerating all the program paths in the CFG, explicitly. Although, most of the path based methods can generate precise results but they are not very scalable, since most of the real-life program have a huge number of paths. In-fact, the number of paths in a program can be exponentially large, even for programs which are guaranteed to terminate. For instance, a program with n branches can have a total of 2^n paths. For a large value of n, the task of determining the longest path can be very complex and time-consuming. But it is possible that a fair share of these 2^n paths, are infeasible in any concrete execution. Therefore, the search space for find the longest path can be reduced with the help of efficient heuristics. Example of path based WCET calculation can be found in [37] and [38].

Tree-based Based Methods , estimate the WCET of a program by performing a bottom-up traversal of the program parse tree. The study by [32], [24] are good examples of tree based method. In most of the tree based methods rules are defined for each type of compound statement in the program parse tree. The WCET of a program is calculated by combining the WCET of each compound statements in the parse tree. Although tree based methods are easy to perform but they have some serious limitation due which they haven't been used widely. On such limitation is the inability to handle flow related information, such as the one proposed by [39]. This implies that the computation can not handle dependencies across different statements. Another limitation with this class of methods is that rules are context-independent making the results of the analysis imprecise.

Implicit Path Enumeration Based Methods , based methods do not enumerate all the program paths explicitly. The approach of IPET based performance estimation was first proposed by [14]. In an IPET based approach, program flow information, as well functional constraints are encoded are linear constraints. The basic idea behind the IPET approach can be described as follows. Each basic block in the program CFG is associated with an execution count variable and cost variable. An execution count variable (x_i), represents the number of times the basic block i has been executed. A constant (c_i), represents the execution time of that basic block. c_i can obtained by various static or measurement based methods. The total execution time of the program can be calculated as

$$\text{Total Execution Time} = \sum_{i=1}^{N} c_i.x_i$$

N denotes the total number of basic blocks in the CFG. Since a basic block can be executed only an integral number of times therefore the variable x_i can only have integer values.



$$x_1 = d_{01} = d_{12} + d_{13} \quad (1)$$
$$x_2 = d_{12} = d_{24} \quad (2)$$
$$x_3 = d_{13} = d_{34} \quad (3)$$
$$x_4 = d_{24} + d_{34} \quad (4)$$
$$d_{01} = 1 \quad (5)$$

Figure 5: Flow constraints to be used in ILP formulation

Figure 5 shows an example control flow graph for a simple if-else program. The edges between the basic blocks $Block_i$ and $Block_j$ are denoted by d_{ij} and the cost of executing a basic block $Block_i$ is denoted by c_i Once all the constraints of the program are specified as linear constraints, an integer linear program (ILP) solver such as CPLEX, can be used for the solving it. In order to obtain the WCET of the program, total execution time of the program has to be maximized, subjected to all structural and functional constraints. For instance, in the example of Figure 5 $x_1 = 1, x_2 = 1, x_3 = 0$ and $x_4 = 1$ maximizes the execution time. The advantage of using IPET based methods for estimate calculation is that, all information such as loop bounds, infeasible paths and user provided constraints can be easily encoded as linear constraints. A limitation of using IPET based methods comes from the fact that ILP problems are usually NP-hard to solve. Therefore finding a solution for a large ILP problem can be very time consuming or even infeasible. Such behaviour can be observed in the experiments by [16], where they perform cache analysis using IPET based method. In the experiment, when associativity of caches is increased, the analysis time increases dramatically. Nevertheless, IPET based approach is an elegant way to represent all program related information in a homogeneous way. Subsequent research works such as that of [15] (for instruction caches) have used scalable approaches such as abstract interpretation for micro-architectural modelling and ILP based approach for bound calculation. Another limitation with the IPET based methods is that, unlike the path based methods it does not output the path which causes the worst case execution time, instead, it just generates the worst case execution time of the program.

2.6 PRECISE MICRO-ARCHITECTURAL MODELING FOR WCET ANALYSIS VIA AI+SAT

Micro-architectural modeling systematically considers the timing effects of underlying hardware platform (*e.g.* pipeline, caches, branch predictors) and it produces the WCET of each basic block in the program control flow graph (CFG). On the other hand, program flow analysis usually involves finding infeasible program paths in the CFG. Such infeasible program paths are ignored during WCET calculation phase to produce a tighter WCET estimate. This WCET analysis process is shown in Figure 6. A crucial observation from Figure 6 is that the micro-architectural modeling and program flow analysis are performed *independently*. As a result, the information computed by program flow analysis is typically *not used* by the micro-architectural modeling. In the absence of program flow information, micro-architectural modeling involves considering many *infeasible* micro-architectural states, which may lead to the imprecision in WCET estimate for each basic block. A typical example of such infeasible micro-architectural state would be the set of memory blocks inserted into the cache along some infeasible program path. With the help of some program flow information (which is already computed during the program flow analysis), such infeasible micro-architectural states can be ignored, which in turn will lead to a tighter WCET of each basic block after the micro-architectural modeling. As a result, the integration of program flow information into micro-architectural modeling may lead to a tighter WCET estimate of the overall program. This is the key idea in this work. The main novelty of our work lies in the consideration of infeasible program paths (computed by program flow analysis) into micro-architectural modeling. Therefore, our work establishes this missing link (in Figure 6) between program flow analysis and micro-architectural modeling.

However, considering program flow information into micro-architectural modeling leads to several technical challenges. A naive strategy is to employ *fully path sensitive* micro-architectural modeling. Such an approach will be infeasible in practice due to the classic

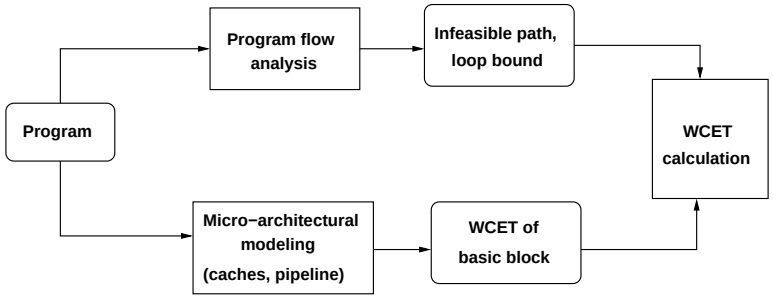


Figure 6: A typical WCET analysis framework

state-space explosion problem [40]. Therefore, micro-architectural modeling for real-time systems is usually accomplished by abstract interpretation. Abstract interpretation is usually efficient but often imprecise. This happens due to the "join" of several micro-architectural states at control flow merge points, which may eventually lead to several infeasible micro-architectural states (*e.g.* infeasible cache contents for cache analysis). However, due to this "join" operation, abstract interpretation is *path insensitive*, which in turn leads to its scalability.

In this section, we propose a generic extension to abstract interpretation (AI) based analysis framework using satisfiability (SAT) checking. Our baseline analysis is abstract interpretation. At any program point, our proposed framework tracks a *partial path* with each micro-architectural state μ. This partial path captures a subset of all the control flow edges along which the micro-architectural state μ has been propagated. We define the partial path as a propositional logic formula φ over the propositions associated with each control flow edge. At each control flow branch (*i.e.* conditional statements), this partial path formula φ is sent to an *oracle*. The *oracle*, in turn, is generated after program flow analysis and checks the *infeasibility* of the partial path (defined by φ). Such a checking can be accomplished by making an *on-the-fly* call to a satisfiability solver (*e.g.* using Minisat [41]). If the partial path was infeasible, its associated micro-architectural state can be ignored for further consideration. Due to a significant progress in SAT solver technologies for the past few decades, such calls to SAT solvers can be processed very efficiently. The set of micro-architectural states generated by our framework is always *tractable*. To control the number of micro-architectural states, we employ strategies to merge different micro-architectural states at appropriate program points. The growth in the number of micro-architectural states (compared to the original abstract interpretation based framework) is always bounded by a *magnitude* equal to the incoming degree of a control flow node, typically a small number. Therefore, we provide a comprehensive and tractable strategy to integrate program flow analysis into micro-architectural modeling.

2.6.1 Overview

In this section, we shall illustrate the central idea behind our approach through a simple example. Through the example, we shall show how the precision of cache analysis can be improved using our proposed framework.

Figure 7(a) shows the control flow graph (CFG) of a program. The label inside each basic block captures the memory blocks accessed by the same basic block. The branch condition is shown beside each conditional branches. For the sake of illustration, let us assume that variable x (used in the conditional branches) is not modified anywhere in the CFG. Additionally, we assume a 2-way set associative cache, where the memory blocks in the CFG are mapped to different cache sets as follows: $m1 \mapsto \mathcal{S}_1$, $m2 \mapsto \mathcal{S}_1$, $m3 \mapsto \mathcal{S}_1$,

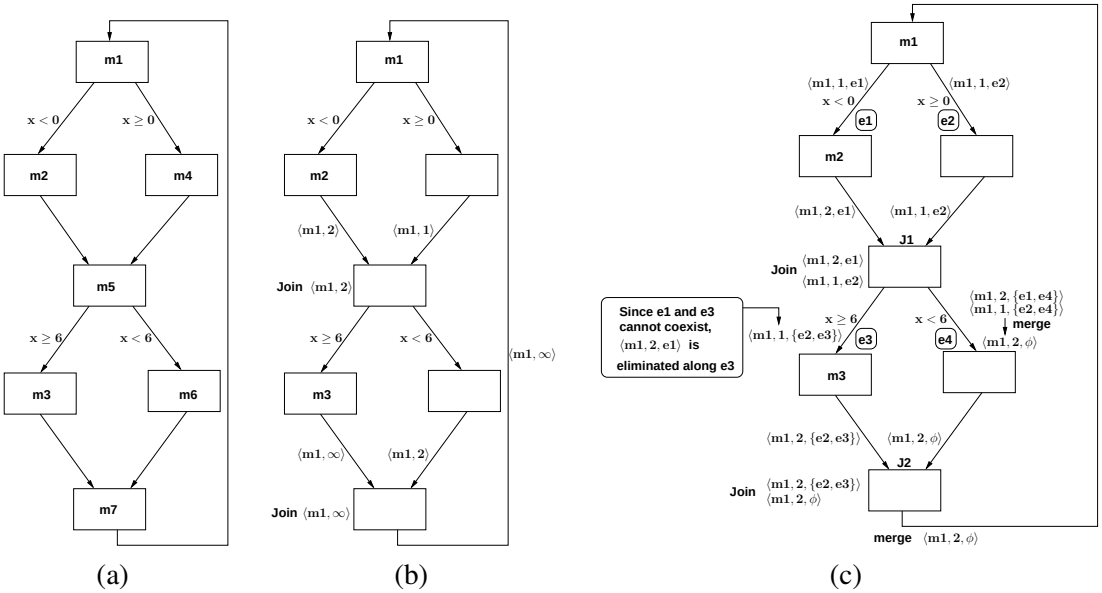


Figure 7: Illustrative example (a) control flow graph with accessed memory blocks shown inside each basic block. (b) original must cache analysis, (c) must cache analysis instantiated by our framework

$m4 \mapsto \mathcal{S}_2$, $m5 \mapsto \mathcal{S}_3$, $m6 \mapsto \mathcal{S}_4$ and $m7 \mapsto \mathcal{S}_7$. $\mathcal{S}_i$ captures the different cache sets. Therefore, in our example, only the memory blocks $m1, m2$ and $m3$ conflict in the cache.

Figure 7(b) shows the state-of-the-art must cache analysis [15] for LRU replacement policy. Each element in the cache state is represented as $\langle m, a \rangle$, where m is the memory block with LRU age a. Let us see the propagation of abstract cache states associated with memory block $m1$. Since $m2$ conflicts with $m1$, the *join* operation at the first control flow merge point computes $\langle m1, 2 \rangle$ (by taking the must join of $\langle m1, 2 \rangle$ and $\langle m1, 1 \rangle$). Since $m3$ also conflicts with $m1$, the control flow after accessing $m3$ will evict $m1$ from the abstract cache state (captured by the element $\langle m1, \infty \rangle$ in Figure 7(b)). As a result, the *join* operation at the second control flow merge computes $\langle m1, \infty \rangle$. Therefore, must analysis cannot conclude any subsequent accesses to $m1$ as *cache hits*. However, careful examination reveals that $x < 0$ and $x \geq 6$ cannot be satisfied for any execution (recall that we assume x is not modified anywhere in the CFG). The must cache analysis was unaware of this infeasible execution. As a result, the traditional must cache analysis assumes two cache conflicts to $m1$ (from $m2$ and $m3$), whereas at most one cache conflict is possible for *any feasible execution*. To summarize, in the absence of any program flow information, abstract interpretation based cache analysis cannot determine that accesses to memory block $m1$ are cache hits (excluding the cold cache miss).

To resolve the gap between program flow analysis and micro-architectural modeling, we propose to extend the abstract domain of the micro-architectural state with partial path information. The instantiation of our proposed framework for cache analysis is shown in Figure 7(c). We first label the control flow edges (as shown by $e1, e2, e3$ and $e4$ in Figure 7(c)) and define a predicate associated with each such labelling. Let us assume $pred_e$ captures the predicate associated with label e. $pred_e$ is *true* if and only if control flow edge e is executed. Program flow analysis can produce infeasible branch pairs in a program (*e.g.* using [42, 43]). For our example, the infeasible branch pairs (*i.e.* $x < 0$ and $x \geq 6$) are captured by the following formula:

$$\Psi_{inf} \equiv \neg pred_{e1} \vee \neg pred_{e3} \tag{6}$$

With the augmented abstract domain, our analysis now labels the cache states with control flow information. Such a labeling enables us to distinguish the control flow along which a single cache state is propagated. As an example, in the beginning, $\langle m1, 1, e1\rangle$ and $\langle m1, 1, e2\rangle$ are propagated along control flow edges $e1$ and $e2$, respectively.

The *join* operation at first control flow merge point keeps both the elements ($\langle m1, 2, e1\rangle$ and $\langle m1, 1, e2\rangle$) in the abstract cache state since they come along different control flows $e1$ and $e2$. The crucial difference is, however, made at the branch point $x \geq 6$. Let us assume that we want to propagate the cache state produced after the first join operation along the control flow labelled $e3$. While propagating a cache state along a branch edge, our framework checks the feasibility of the cache state along the same branch. Therefore, when we try to propagate $\langle m1, 2, e1\rangle$ along $e3$, we check the feasibility of the state along $e3$ by consulting the information generated by program flow analysis (*i.e.* Ψ_{inf}). We make a call to the SAT solver to check the feasibility of the following formula:

$$\begin{aligned} \varphi \quad &\equiv \Psi_{inf} \wedge pred_{e1} \wedge pred_{e3} \\ &\equiv (\neg pred_{e1} \vee \neg pred_{e3}) \wedge pred_{e1} \wedge pred_{e3} \end{aligned} \tag{7}$$

This is due to the fact that the propagation of $\langle m1, 2, e1\rangle$ along $e3$ must execute both the control flow edges $e1$ and $e3$, which in turn means that $pred_{e1} \wedge pred_{e3}$ must be *true*. Since the formula in Equation 7 is *unsatisfiable*, we do not propagate the cache state $\langle m1, 2, e1\rangle$ along $e3$. On the other hand, since $\Psi_{inf} \wedge pred_{e2} \wedge pred_{e3}$ is satisfiable, $\langle m1, 1, e2\rangle$ is propagated along $e3$. Additionally, such a propagation of cache state $\langle m1, 1, e2\rangle$ along $e3$ updates the control flow information of the cache state from $e2$ to $\{e2, e3\}$ (as shown by the element $\langle m1, 1, \{e2, e3\}\rangle$ in Figure 7(c)). $\langle m1, 1, \{e2, e3\}\rangle$ now captures that the original cache state $\langle m1, 1\rangle$ has been propagated through control flow edges $\{e2, e3\}$.

Now let us consider the branch edge labelled $e4$. We found that both the formula $\Psi_{inf} \wedge pred_{e1} \wedge pred_{e4}$ and $\Psi_{inf} \wedge pred_{e2} \wedge pred_{e4}$ are *satisfiable*. Therefore, both the cache states $\langle m1, 2, e1\rangle$ and $\langle m1, 1, e2\rangle$ are propagated along $e4$ and the control flow information of $\langle m1, 2, e1\rangle$ and $\langle m1, 1, e2\rangle$ are updated to $\{e1, e4\}$ and $\{e2, e4\}$, respectively (as shown by $\langle m1, 2, \{e1, e4\}\rangle$ and $\langle m1, 1, \{e2, e4\}\rangle$ in Figure 7(c)). To control the number of cache states containing $m1$, we perform a merge operation at control flow edge $e4$. The merge operation for must cache analysis takes the *maximum* age of a memory block and loses the entire control flow information. As a result, after merging $\langle m1, 2, \{e1, e4\}\rangle$ and $\langle m1, 1, \{e2, e4\}\rangle$ for must cache analysis, we get $\langle m1, 2, \phi\rangle$. Due to this merge operation, we can always control the number of micro-architectural states in our framework.

It is worthwhile to note the difference between merge and join operation in our framework. For the time being assume that we had performed the *merge* operation between $\langle m1, 2, \{e1\}\rangle$ and $\langle m1, 1, \{e2\}\rangle$ at the first control flow join point (*i.e.* at $J1$). As a result, we had obtained a cache state $\langle m1, 2, \phi\rangle$ after the first control flow join. If we try to propagate the cache state $\langle m1, 2, \phi\rangle$ along $e3$, we check the satisfiability of a formula $\Psi_{inf} \wedge pred_{e3}$, which is clearly *satisfiable*. Consequently, we had not eliminated any cache state along $e3$ and all subsequent accesses to $m1$ would not be classified as *cache hits*. Continuing in a similar sequence of join and merge operation, we obtain the cache state $\langle m1, 2, \phi\rangle$ propagated along the backedge. As a result, all subsequent accesses of $m1$ can be categorized as *cache hits*. Note that the key difference in our framework was made by removing the infeasible cache state $\langle m1, 2\rangle$ (in the traditional must cache analysis framework) along control flow edge $e3$. This infeasible cache state was detected using the infeasible path information computed by program flow analysis (*i.e.* Ψ_{inf}) and augmenting the abstract interpretation framework.

2.6.2 General Framework

In this section, we shall introduce the general idea behind our analysis framework. We shall show how an abstract interpretation based analysis framework can be augmented with the help of a satisfiability solver to generate more precise analysis outcome.

PROGRAM FLOW ANALYSIS Our proposed framework uses program flow analysis to rule out certain infeasible micro-architectural states. For program flow analysis, we currently look at finding the *infeasible branch pairs*. Assume two branch conditions $x \leq 0$ and $x \geq 2$ in a program. If x is not modified, both $x \leq 0$ and $x \geq 2$ cannot be *true* for any execution. As a result, the control flow edges associated with the *true* evaluations of $x \leq 0$ and $x \geq 2$ constitute an infeasible branch pair. Such infeasible branch pairs can be computed automatically (such as using [42], [43]) or they can be provided manually by the user.

In the past few decades, satisfiability (SAT) solver technology has made significant progress. An interesting feature about infeasible branch pairs is that they can easily be encoded as propositional logic formula. Let us introduce an atomic proposition $pred_e$ associated with each control flow edge e in the program. $pred_e$ captures the execution of control flow edge e. $pred_e$ is *true* if control flow edge e is executed and *false* otherwise. Therefore, an infeasible branch pair $\langle b_i, b_j \rangle$ can be encoded as the following propositional formula:

$$\varphi \equiv \neg pred_{b_i} \vee \neg pred_{b_j} \tag{8}$$

At the end of program flow analysis, therefore, we have a set of clauses (as shown in Equation 8) in *conjunctive normal form* (CNF). Let us assume Ψ_{inf} represents this CNF formula. Therefore, Ψ_{inf} captures certain infeasible path patterns (specifically infeasible branch pairs) in a program.

Given the information computed by program flow analysis (*i.e.* Ψ_{inf}), we define an oracle Θ on any propositional formula η as follows:

$$\Theta(\eta) = \begin{cases} true, & \text{if } \Psi_{inf} \wedge \eta \text{ is satisfiable;} \\ false, & \text{otherwise} \end{cases} \tag{9}$$

Note that any satisfiability solver (such as Minisat [41]) can be used as the oracle Θ. Θ will be used to eliminate infeasible micro-architectural states.

2.6.3 Augmenting Abstract Interpretation

The key idea behind our analysis framework is to inject the notion of path sensitivity inside abstract interpretation. A fully path sensitive approach is definitely not scalable, as it leads to a path explosion. Therefore, we augment the abstract interpretation in a fashion such that path explosion never occur and the state space of the analysis can always be controlled. In the following, we shall briefly describe the key components of the analysis.

CHANGING THE ABSTRACT DOMAIN Assume an abstract interpretation based analysis framework with abstract domain $\mathbb{D}$. To handle partial path sensitivity, our proposed analysis framework augments this abstract domain $\mathbb{D}$ as follows:

$$\mathbb{D}' : \mathbb{D} \times \mathcal{P}(\mathbb{E}) \tag{10}$$

In the above equation, $\mathbb{E}$ captures the set of all control flow edges in the program and $\mathcal{P}(\mathbb{E})$ represents the set of all subsets of $\mathbb{E}$. Therefore, at any specific program point p, an element

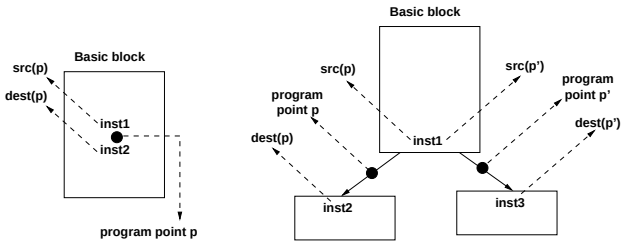


Figure 8: Program point (a) inside a basic block, (b) at a branch location

from the changed abstract domain $\mathbb{D}'$ is of the form $\langle d, \{e_1, e_2, \ldots, e_k\}\rangle$ where $d \in \mathbb{D}$ and $\{e_1, e_2, \ldots, e_k\}$ captures the set of control flow edges along which d has been propagated to p.

TRANSFER AND JOIN FUNCTION Before going into the details of *transfer* and *join* function, we first briefly describe the notion of *program points* in our analysis framework. We define the program point as the control flow between two instructions. We distinguish the two different types of program points in our framework as shown in Figure 8. Figure 8(a) captures the control flow inside a basic block of the program CFG. On the other hand, Figure 8(b) captures the control flow between two different basic blocks. For a specific program point p, we shall use $src(p)$ to denote the *source* of the control flow captured by p, whereas $dest(p)$ will be used to denote the destination of the control flow captured by p.

Transfer function of our proposed analysis framework is applied at each program point. However, our proposed transfer function has two key components, depending on the location of the program point. More precisely, our proposed transfer function has the following semantics ($\mathbb{P}$ denotes the set of all program points):

$$\tau : \mathbb{D}' \times \mathbb{P} \rightarrow \mathbb{D}'$$

$$\tau(d', p) = \begin{cases} \tau_{br} \bullet \tau_{in}(d', p), & \text{if } src(p) \text{ is at the end} \\ & \text{of a basic block;} \\ \tau_{in}(d', p), & \text{otherwise.} \end{cases} \tag{11}$$

$\bullet$ denotes the function composition. In Equation 11, τ_{br} captures the transfer function used at the control flow across basic blocks (Figure 8(b)) and τ_{in} captures the transfer function used at the control flow inside a basic block (Figure 8(a)). The computation of τ_{in} largely depends on the type of analysis, as it requires updating the abstract state by considering each instruction. τ_{br}, on the other hand, is the key to our proposed framework and it is used to eliminate the *spurious* (*i.e.* infeasible) abstract states. Assume that e_p captures the control flow edge associated with program point p when $src(p)$ is at the end of a basic block. Further assume $d' \in \mathbb{D}'$ and $d' = \langle d, E\rangle$ (where $E \subseteq \mathbb{E}$ and $\mathbb{E}$ is the set of all control flow edges). We define $\tau_{br}(d', p)$ as follows:

$$\tau_{br}(d', p) = \begin{cases} \phi, & \text{if } \Theta(\bigwedge_{e \in E} pred_e \wedge pred_{e_p}) \text{ is } false \\ \langle d, E \cup \{e_p\}\rangle, & \text{otherwise.} \end{cases} \tag{12}$$

Recall that $pred_e$ denotes the atomic proposition which evaluates to *true* if and only if control flow edge e is executed. Θ is the oracle (as described in Equation 9) used to check the feasibility of control flow $E \cup \{e_p\}$. Equation 12 serves two purposes: first, to eliminate

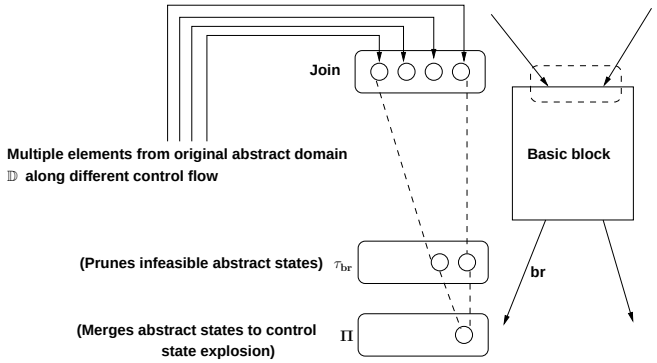


Figure 9: A schematic representation of the *join*, τ_{br} and merge (Π) used in our proposed analysis framework

spurious abstract states (captured by the first case in Equation 12) and secondly, to associate the notion of path sensitivity with abstract states (captured by the second case in Equation 12).

Join operation with our augmented abstract domain $\mathbb{D}'$ operates in a similar fashion as with the *join* operation with original abstract domain $\mathbb{D}$, with one crucial difference. After the *join* operation is performed with $\mathbb{D}'$, a single element from the original abstract domain $\mathbb{D}$ may have multiple instances in the *joined* abstract state. These multiple instances may appear due to the propagation of a single element in $\mathbb{D}$ along different control flow paths. As a result, a single element from the original abstract domain $\mathbb{D}$ might be associated with different subsets of control flow edges, leading to different elements in the augmented abstract domain $\mathbb{D}'$ in the joined abstract state.

CONTROLLING THE NUMBER OF ABSTRACT STATES The *join* operation in our proposed framework may enlarge the number of abstract states compared to the original abstract interpretation based framework. As a result, performing the join operation in an uncontrolled fashion may lead to state explosion. Therefore, our proposed framework controls the number of elements in the abstract state at each control flow edges (*i.e.* after performing the τ_{br} operation). This is done by pruning the number of tractable elements with a special operation Π. If there is a pair of elements $d'_1, d'_2 \in \mathbb{D}'$ such that $d'_1 = \langle d, E_1 \rangle$ and $d'_2 = \langle d, E_2 \rangle$, they are merged to a single element $\langle d, \phi \rangle$ using the operation Π. In an abstract state, Π is applied until there is *at most* one element in the abstract state ($\in \mathbb{D}'$) for each element in the original abstract domain $\mathbb{D}$.

Figure 9 shows a schematic view of our overall framework. The *join* operation used by our framework may increase the number of elements in the abstract state, due to the presence of different control flows. τ_{br} operation at a branch location may prune some of the infeasible elements in an abstract state as also shown in Figure 9. Finally, after merge operation (Π), the size of the abstract state is controlled, which in turn lead to a tractable analysis framework. Note that we apply the merge operation at each control flow edge. Therefore, the expansion in the number of elements after the *join* operation is bounded by a magnitude equal to the incoming degree of any basic block, typically a small number.

2.6.4 Instruction Cache Analysis via AI+SAT

In this section, we shall instantiate our proposed framework for instruction cache analysis. Abstract interpretation based instruction cache analysis was initiated in [15]. Each instruction is categorized as *all-hit* (AH), *all-miss* (AM) or *not-classified* (NC). An instruction is categorized as AH, if it is found in the cache whenever it is accessed, whereas an instruction is

categorized AM, if it is never in the cache whenever it is accessed. If an instruction cannot be categorized as either AH or AM, it remains *not-classified* (NC). For such a categorization of different instructions, usually two different analyses are employed: *must* and *may*. *Must* cache analysis is used for statically predicting a *sound* under-estimation of cache content at each program point. As a result, must cache analysis can be used for AH categorization of different instructions. On the other hand, *may* cache analysis is used for statically predicting a *sound* over-estimation of cache content at each program point. Consequently, we can use *may* cache analysis for AM categorization of instructions.

MODIFYING ABSTRACT DOMAIN The abstract domain of the original instruction cache analysis can be defined as a cross product of two different abstract domains as follows:

$$\mathbb{D} : \mathcal{P}(\mathbb{M} \times \mathbb{A}) \tag{13}$$

with

$$\mathbb{A} = \{1, \ldots, K\} \cup \{\infty\} \tag{14}$$

where $\mathbb{M}$ denotes the set of all memory blocks and $\mathbb{A}$ denotes the set of all possible *ages* of a memory block inside a K-way set-associative cache. A tuple of the form $\langle m, \infty \rangle \in \mathbb{D}$ captures that m does not reside in the cache.

In our proposed framework we augment the original abstract domain $\mathbb{D}$ to $\mathbb{D}'$ as follows:

$$\mathbb{D}' : \mathcal{P}(\mathbb{M} \times \mathbb{A} \times \mathcal{P}(\mathbb{E})) \tag{15}$$

where $\mathbb{E}$ is the set of all control flow edges. Therefore, an entity in the abstract domain $\mathbb{D}'$ is a triplet $\langle m, a, E \rangle$.

MODIFYING TRANSFER AND JOIN FUNCTION The modified transfer function can now be described as follows:

$$\tau : \mathbb{D}' \times \mathbb{P} \rightarrow \mathbb{D}'$$

$$\tau(\langle m, a, E \rangle, p) = \begin{cases} \tau_{br} \bullet \tau_{in}(\langle m, a, E \rangle, p), \text{ if } src(p) \text{ is at the end} \\ \qquad\qquad \text{of a basic block;} \\ \tau_{in}(\langle m, a, E \rangle, p), \text{otherwise} \end{cases} \tag{16}$$

$\bullet$ denotes the function composition and the transfer function τ now has two different controls depending on the location of program point p. If we assume that m_p denotes the memory block accessed at $src(p)$, τ_{in} can be defined as follows:

$$\tau_{in}(\langle m, a, E \rangle, p) = \begin{cases} \langle \mathcal{U}_{repl}(\langle m, a \rangle, p), \phi \rangle, & \text{if } m_p = m \\ \langle \mathcal{U}_{repl}(\langle m, a \rangle, p), E \rangle, & \text{otherwise} \end{cases} \tag{17}$$

In Equation 17, $\mathcal{U}_{repl}$ captures the instruction cache update operation (in the original abstract domain $\mathbb{D}$) for a particular cache replacement policy $repl$. Note that our framework is not restricted to a particular replacement policy employed by the instruction cache.

τ_{br} is applied at a branch point. Some of the triplets present in the input abstract cache state may not be feasible along some branches. Therefore, while performing the transfer operation along a control flow edge, only the feasible triplets are transferred to the output abstract cache state. Assume that e_p captures the control flow edge associated with program point p. We define τ_{br} for instruction cache analysis as follows:

$$\tau_{br}(\langle m, a, E \rangle, p) = \begin{cases} \emptyset, \text{if } \Theta(\bigwedge_{e \in E} pred_e \wedge pred_{e_p}) \text{ is } false \\ \langle m, a, E \cup \{e_p\} \rangle, \text{otherwise} \end{cases} \tag{18}$$

Recall that Θ is the oracle (as described in Equation 9) used to check the feasibility of control flow $E \cup \{e_p\}$. While performing a Join operation, a memory block m, may appear along different control flows. Moreover, m might have different *ages* in the cache along different control flows. In our proposed framework, we do not lose the information about the different ages of the same memory block along different control flows. Therefore, in our proposed framework, we define the *must* and *may* join operations as follows:

$$\bigsqcup_{Must}, \bigsqcup_{May} : \mathbb{D}' \times \mathbb{D}' \rightarrow \mathbb{D}' \tag{19}$$

$$\bigsqcup_{Must}(D_1, D_2) = \{\langle m, a_1, E_1\rangle \in D_1 \mid \exists a_2, E_2 : \langle m, a_2, E_2\rangle \in D_2\} \cup \{\langle m, a_2, E_2\rangle \in D_2 \mid \exists a_1, E_1 : \langle m, a_1, E_1\rangle \in D_1\} \tag{20}$$

$$\bigsqcup_{May}(D_1, D_2) = D_1 \cup D_2 \tag{21}$$

May join operation simply takes the union of two abstract cache states, on the other hand, must join operation takes also the union operation, but restricted to the memory blocks which are present in both the abstract cache states (D_1 and D_2). It is important to note that the different *ages* ($\in \mathbb{A}$) of a memory block (along different control flows) will be retained after Equations 20-21. Changes in the *age* of a memory block in a cache set are handled in merge operation (discussed next).

MERGING ABSTRACT STATES We shall now show how we control the number of abstract cache states at each control flow edges. Note that the join operation in our proposed framework leads to more elements in the abstract cache state as compared to the original analysis proposed in [15]. However, since we prune the number of abstract cache states at each control flow edges, the expansion in the number of abstract cache states is still bounded.

Assume that we obtain an abstract cache state $D \in \mathbb{D}'$ after performing τ_{br} at a branch location. The output of merge operation depends on the type of cache analysis (*i.e. must* or *may* cache analysis). Assume that Π_{must} (Π_{may}) denotes the merge operation applied for *must* (*may*) cache analysis. The main purpose of the merge operation is to control the number of cache states and therefore, after each merge operation, we ensure that the resulting abstract cache state contain *at most one* element for each memory block. Consider two elements $\langle m, a_1, E_1\rangle, \langle m, a_2, E_2\rangle \in D$. The output of Π_{must} and Π_{may} will be as follows:

$$\Pi_{must}(\langle m, a_1, E_1\rangle, \langle m, a_2, E_2\rangle) = \langle m, \max(a_1, a_2), \phi\rangle \tag{22}$$

$$\Pi_{may}(\langle m, a_1, E_1\rangle, \langle m, a_2, E_2\rangle) = \langle m, \min(a_1, a_2), \phi\rangle \tag{23}$$

Therefore, after performing Π_{must}, we retain the *maximum age* of each memory block m. On the other hand, after performing Π_{may}, we retain the *minimum age* of each memory block m. Since the merge operation is performed at each control flow edge, we can now state the following property for our proposed framework:

Property 2.6.1 *Let us assume $\mathbb{B}$ denotes the set of all basic blocks and $deg_{in}(B)$ denotes the incoming degree of a basic block B in the CFG. For a K-way set-associative cache, there could be at most $|K| \times \max_{B \in \mathbb{B}} deg_{in}(B)$ number of elements for any memory block in any abstract cache state.*

Since both $deg_{in}(B)$ and K are typically small, we can easily control the number of abstract cache states.

It is worthwhile to mention that the necessary pruning of abstract cache states has already been performed during the computation of τ_{br} (Equation 18). Therefore, by merging the cache states using Π_{must} and Π_{may} we are not entirely losing the partial path sensitivity used by our framework. On the other hand, merging of abstract cache states leads our computation to be tractable in practice.

Since our proposed framework is built up on the basic abstract interpretation (AI) approach, it is guaranteed to give at least as precise cache analysis as the basic AI approach [15]. The main purpose of our proposed framework is to integrate the program flow information into cache analysis. With little or absence of any program flow information, our framework will give exactly same result as in [15].

2.6.5 Data Cache Analysis

In this section, we describe the extension of our framework for data cache analysis. We use the *scope-aware persistent* (SCP) analysis [23] as the baseline analysis for data caches. SCP analysis was selected because the WCET estimates generated by this method are *safe* and more accurate than other existing methods. In SCP analysis, each memory block m is associated with a temporal scope. For a particular memory block m, its temporal scope denotes the set of loop iterations where m might be accessed. If two different memory blocks map to the same cache set but they have disjoint temporal scopes, they cannot conflict in the cache. SCP analysis categorizes data blocks as persistent or non-persistent, on the basis of their temporal scopes. Although the SCP analysis is more accurate than other abstract interpretation based methods, the WCET estimated by the method can still be over-estimated. This might happen due to the presence of infeasible paths in the program CFG. By extending the SCP analysis using our framework, we can remove such over-estimation and thereby produce a more accurate WCET. Since the SCP analysis is based on temporal scopes, the transfer functions (*i.e.* data cache update operations) are defined with respect to each program loop. For a given loop level L, the transfer function of our proposed framework is similar to Equation 16 and it can be described as follows:

$$\tau(\langle m,a,E\rangle,p,L) = \begin{cases} \tau_{br} \bullet \tau_{in}(\langle m,a,E\rangle,p,L), \\ src(p) \text{ is at the end of a basic block;} \\ \\ \tau_{in}(\langle m,a,E\rangle,p,L), \text{otherwise} \end{cases} \tag{24}$$

Note that $\bullet$ denotes function composition. τ_{br} for the extended SCP analysis operates in a similar fashion as in the instruction cache analysis (see Equation 18). However, τ_{in} for the SCP analysis is slightly different from the τ_{in} described in Equation 17. This is due to the fact that SCP analysis is scope-sensitive (unlike the instruction cache analysis). Let us assume $\mathbb{M}_p$ denotes the set of memory blocks accessed at a program point p. For a given loop level L, τ_{in} can be defined as follows: $\tau_{in}(\langle m,a,E\rangle,p,L) =$

$$\begin{cases} \langle m,a,E\rangle, \text{ if } \forall m_i \in \mathbb{M}_p.\ (\psi(m_i) \cap \psi(m) = \phi \vee \\ \qquad\qquad \pi(m_i) \neq \pi(m)) \\ \langle \mathcal{UD}(\langle m,a\rangle,p,L),E\rangle, \text{ if } \exists m_i \in \mathbb{M}_p.\ (\psi(m_i) \cap \psi(m) \neq \phi \\ \qquad\qquad \wedge \pi(m_i) = \pi(m)) \wedge m \notin \mathbb{M}_p \\ \langle \mathcal{UD}(\langle m,a\rangle,p,L),\phi\rangle, \text{ otherwise} \end{cases} \tag{25}$$

$\psi(m)$ denotes the set of iterations in loop L where m might be accessed (*i.e.* temporal scope of m) and $\pi(m)$ captures the cache set in which memory block m is mapped. The first case (in Equation 25) captures the scenario when none of the memory blocks in $\mathbb{M}_p$ conflict with m in the data cache (either due to disjoint temporal scopes or due to the mapping in disjoint cache sets). If some memory block in $\mathbb{M}_p$ conflicts with m in the data cache, the scope-aware data cache update operation $\mathcal{UD}$ (used in [23]) is used to update the data cache state. Since data cache replacement policy may only change $\mathcal{UD}$ in Equation 25, our proposed framework remains unchanged for different data cache replacement policies. The SCP join function as defined in [23], can be modified in the same fashion as we do for the join operation in the instruction cache analysis (see Equations 20-21).

2.6.6 Branch Target Buffer Analysis

Branch Target Buffer (BTB) is used to predict the target address of a branch instruction, before the target address is actually computed. Since the computation of a target address has some associated penalty, correct prediction of a target address from BTB may greatly improve the program execution time. As a result, an improved BTB analysis may lead to a more accurate WCET prediction. Previous study (*e.g.* [32]) has shown the importance of BTB analysis for static WCET prediction. The work of [32] uses an abstract interpretation based framework for statically analyzing the BTB content and categorizing the branch instructions on the basis of BTB states. As with any other abstract interpretation based approach, the analysis proposed in [32] is also path in-sensitive and it suffers due to the estimation of infeasible BTB states. We extend the framework for BTB analysis as proposed in [32] with our approach. The new domain of the BTB analysis can be formulated as follows:

$$\mathbb{D}' : \mathcal{P}(\mathbb{BR} \times \mathbb{A} \times \mathcal{P}(\mathbb{E})) \tag{26}$$

where $\mathbb{BR}$ is the set of all branch instructions. All other parameters in the equation have the same interpretation as in equation (15). The join and the transfer functions can be modified in exactly the same fashion as they were modified for the instruction cache analysis (see Equations 16-21).

2.6.7 Shared Instruction Cache Analysis

Finally we show how our framework can be extended for analyzing multi-core systems with shared instruction caches. We use our previous work [44] as a baseline for the augmented abstract interpretation framework. Let us assume a multi-core system where each core has a private L1 cache and all the cores share an L2 cache. The work in [44] first analyses both the L1 and L2 cache using [15, 45] ignoring the inter-core cache conflicts and finally, it recategorizes the memory blocks in L2 cache by taking into account the shared cache conflicts.

Assume a program P_1 running on Core 1 and assume that P_1 accesses a memory block m_{p1}. Further assume m_{p1} is categorized as AH in the shared L2 cache without considering inter-core cache conflicts. After inter-core cache conflict analysis, the AH categorization of memory block m_{p1} is changed to NC if and only if the following condition holds: $k - age_{singlecore}(m_{p1}) \leq X$, where k is the associativity of the shared L2 cache, $age_{singlecore}(m_{p1})$ captures the *age* of memory block m_{p1} in the shared L2 cache set before inter-core conflict analysis and X is the amount of shared cache conflicts generated by cores other than Core 1.

If we use augmented abstract interpretation instead of basic abstract interpretation, we can get a precise cache hit-miss categorization for L1 cache. A precise cache hit-miss

categorization of memory blocks in L1 cache leads to a lesser number of memory blocks accessing the shared L2 cache. As a result, the amount of shared cache conflicts (*i.e.* X) may reduce. Moreover, since our proposed framework may generate a better must cache analysis (see Section 2.6.4), it may also reduce the quantity $age_{singlecore}(m_{p1})$. Both of these developments in turn reduce the number of memory blocks of P_1 which need to be re-categorized to NC in the shared L2 cache. As a result, we may increase the accuracy of WCET estimates even in the presence of shared caches.

2.6.8 Experimental Evaluation

EXPERIMENTAL SET-UP The experiments were performed using the timing analyser Chronos [46]. It uses a 5-stage pipeline with in-order execution, when generating the WCET estimates for our experiments. Chronos uses the abstract interpretation based framework proposed in [15], to analyze instruction caches. This serves as the baseline for our instruction cache analysis. The baseline for our data cache analysis framework was published in [23]. The BTB analysis is implemented into Chronos using the abstract interpretation based framework proposed in [32]. The framework proposed in [44] serves as a baseline analysis for the multi-core, shared instruction cache analysis. To check the satisfiability of a given partial path φ, we use the open-source SAT solver Minisat [41].

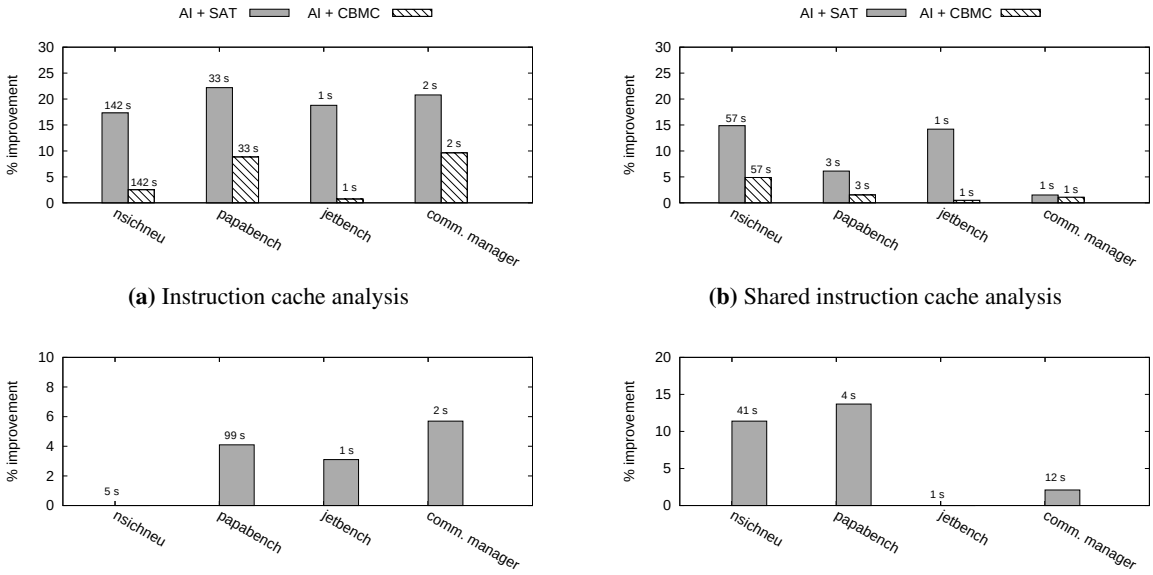
Subject Programs	Description	Code size	
		Bytes[1]	LOC
nsichneu	Simulates an extended Petri Net. Auto-generated code with many if-statements [47]	63720	4253
Papabench	Auto-navigation utility from an Unmanned Aerial Vehicles (UAV) controller [48]	16920	1097
Jetbench	Single-thread *getThermo* utility from a real-time jet engine performance calculator [49]	6984	315
Communication manager	Auto-generated code from the Rhapsody [50] model of CTAS weather manager [51]	4248	273

Table 2: Program Set I

Table 4 shows the subject programs for our experiments. Note that the prime motivation behind our work is to reduce the over-approximation in WCET estimation, due to the presence of infeasible paths. Infeasible paths can often be found in the programs, auto-generated from high level modeling languages. Although, it is not uncommon to have a few infeasible paths in manually written programs as well. Therefore, we choose a combination of auto-generated and manually written programs for our experiments. All of our experiments were performed on a machine having an Intel Core-i5 processor with 4 GB RAM and running Ubuntu 9.04 OS. For all our experimental results, we measure the WCET improvement as $\frac{WCET_{base}-WCET_{augmented}}{WCET_{base}} \times 100\%$, where $WCET_{augmented}$ captures the WCET obtained using our approach and $WCET_{base}$ captures the WCET obtained using the baseline approach.

INSTRUCTION CACHE RESULT Figure 10a shows the results of Instruction cache analysis using the augmented abstract interpretation (AI+SAT) approach. We perform the analysis for all programs in Table I, for a block size of 64 Bytes, on a 4-way set-associative L1 cache. For a fair comparison, we chose the cache size approximately equal to the program size (closest power of two), for each subject program. We assume that there is no L2 cache for this experiment and the latency for memory access is 36 cycles. We also compare our approach

1 code size in bytes = ending instruction address - starting instruction address



(a) Instruction cache analysis

(b) Shared instruction cache analysis

(c) Data cache analysis

(d) Branch target buffer analysis

Figure 10: Improvement in the WCET accuracy via AI+SAT approach, analysis time (in seconds) is shown above each bar

with the work in [18] (say AI+CBMC). The framework proposed in [18] first generates the cache hit-miss categorization of a program using basic abstract interpretation. It then uses CBMC [52] to refine the set of NC categorized memory blocks. As a result of this refinement, the WCET estimate might improve.

We compare the improvements in WCET estimation achieved by the AI+SAT approach, with that of AI+CBMC approach. We observe that the estimates generated by the AI+SAT approach are more accurate than that of the AI+CBMC approach, when both the analysis are run for a comparable amount of time. We observed a maximum improvement in WCET estimates of up to 25% for this set of experiments. This improvement can be attributed to the fact that all the programs in Table I have multiple program paths inside loops. Some of these program paths are infeasible and hence cause some over-estimation in the base analysis. By applying the AI+SAT approach, we were able to remove some of the over-estimation caused due to such infeasible paths. However, it is worthwhile to mention that AI+CBMC is a verification based method. As a result, it might be able to produce better estimates, if run for sufficiently long. This introduces a trade-off between WCET accuracy and analysis time for using AI+SAT over AI+CBMC. In our experiments, we found that AI+SAT could produce a 17% more accurate WCET estimate for `nsichneu` compared to the baseline analysis in approximately 142 seconds, whereas AI+CBMC approach produced a maximum improvement of 30% in approximately 1756 seconds.

DATA CACHE RESULT Figure 10c shows the results of data cache analysis using the AI+SAT approach. Note that the baseline analysis for data cache is scope-aware persistence analysis [23]. We assume that L1 cache hit latency is 1 cycle whereas the memory access latency is 36 cycles. Although all the programs in Table I have very little data accesses on their infeasible paths, we still get a noticeable improvement for most of the programs via AI+SAT approach. The improvements shown in Figure 10c are the maximum improvement for programs in Table I, for any given cache size. The maximum time taken for all the experiments under this section was under 2 minutes. As Figure 10c shows, all the subject programs other than `nsichneu` have noticeable improvement in their WCET estimates. This is because `nsichneu` has a very small data set and it has very little data accesses across its

infeasible paths. However, this should not be considered as a limitation of our approach, as the AI+SAT based framework will give reasonable improvement for programs with many conflicting data accesses along infeasible paths. Another factor which might be affecting the efficacy of our method can be the underlying address analysis. Address analysis usually generates an over-approximation of the memory ranges which can be accessed for a given data access. Due to this over-approximation, additional memory blocks might lose their control flow information over the merge operations. This leads to an imprecision in the abstract cache states and overall WCET. Therefore, using a better address analysis will directly improve the WCET accuracy via our approach.

BRANCH TARGET BUFFER RESULT Figure 10d shows the results of branch target buffer (BTB) analysis, using the AI+SAT approach. All the experiments for this set of analysis took less than a minute to complete. We used a 2-way set associative BTB with 256 entries. Also we took the branch misprediction penalty as 15 cycles. The maximum improvement in WCET estimation was observed for `Papabench` (approximately 14%). We did not observe any considerable improvement for `Jetbench`. This can be due to fact that `Jetbench` has very less branch instructions along the infeasible program paths. For the other two subject programs we observed a moderate improvement in the WCET estimation.

SHARED CACHE RESULT Finally, we present the results for the shared instruction cache analysis. For this set of experiments, we assume a multi-core system with two cores. Moreover, we assume that each core has a private L1 instruction cache and the L2 instruction cache is shared by both the cores. We ran a program from Table I on $Core_1$ and a program from Table II (Table II programs are taken from [47]) on $Core_2$.

Subject programs	Description	Code size	
		Bytes[1]	LOC
jfdcint	Discrete cosine transform on 8x8 block	5512	375
edn	Signal processing application	3160	285
ndes	Complex embedded code	3816	231
adpcm	ADPCM coder	12568	879

Table 3: Program Set II

We used a direct mapped L1 cache with a cache size of 256 Bytes and a block size of 32 bytes. We choose a small L1 cache to generate sufficient number of conflicts in the L2 cache. The size of the shared L2 cache is chosen depending upon the code size of the program running on $Core_1$. L2 cache hit latency is taken as 6 cycles and memory access latency is taken as 30 cycles. We perform the shared cache analysis using the AI+SAT approach and measure the improvement in WCET estimation for the programs running on $Core_2$. We then compare the improvements achieved by our approach, with the improvements achieved by using the AI+CBMC [18] approach (using the same cache configurations). AI+CBMC was run for the same amount of time as AI+SAT.

Figure 10b shows the *geometric mean* of improvements in WCET estimation, for all the programs running on $Core_2$. All of the results reported in Figure 10b were completed under 2 minutes. We observed a noticeable improvement in the WCET estimates for all the programs running on $Core_2$, produced by the AI+SAT approach. We also observed that the improvements achieved by the AI+SAT were significantly better than the improvement achieved by the AI+CBMC, in the same amount of time. By applying AI+SAT to the programs running on $Core_1$, we were able to reduce a reasonable number of conflicts in the shared cache. This leads to more accurate WCET estimates, for programs running on $Core_2$. Using AI+SAT, the maximum improvement was observed for `ndes` (while running `nsichneu`

on the other core), which was around 41% in 50 seconds, whereas for the same experiment, maximum improvement using AI+CBMC was observed to be around 42% in 85 seconds.

2.7 PERFORMANCE-AWARE TEST GENERATION TECHNIQUES

There have been few works that address the need for techniques that can automatically explore programs and generate test-inputs that witness suboptimal performance behaviour. One of our works proposes a technique to address this need and is further described in Chapter 3. However, in the following paragraphs, we shall discuss only two works ([53] and [54]) that are related to this topic.

Computational complexity is one of the factors that affects the execution time of a program. The work of [53] proposes a technique to automatically find test inputs for which the program exhibits worst-case computational complexity. The technique in [53] uses symbolic execution to explore all feasible program paths for a small (the definition of small is ambiguous) input size. While exploring the program for a small inputs size, the technique analyses the executions to create a *branch policy generator*. The generator is constructed such that it contains information pertaining to the worst-case path, in the observed executions. After the generator has been crafted, it is used to guide the symbolic exploration for larger input sizes. It is worthwhile to know that the technique in [53] does not takes the underlying micro-architecture in to account. Therefore, even though this techniques can provide some idea about the influence of an input on the program behaviour but it cannot be used to identify test-inputs that lead to bad performance due to specific micro-architectural components.

The work of [54] presents a technique more relevant to generating test-inputs that highlight suboptimal performance in micro-architectural components. It proposes a technique that uses constraint-based test generation [5, 55] to partition the input domain of a program with respect to cache performance. Once all the partitions are computed, some manual interventions is needed to locate the set of program locations that may exhibit issues related to cache performance. It proposes a way to compute the cache performance range for each such partition. The cache performance range in [54] is computed via *static invariant generation* methods. As a result, the computed cache-performance range may be over-approximated, leading to *false positives*.

2.8 CHAPTER SUMMARY

In this chapter we presented a number of methods that can be used for performance analysis. Specifically, the methods discussed included methods on performance profiling, performance estimation and performance testing. It is worthwhile to know that no one class of method may be suitable for all performance analysis needs. In the scenario which requires the estimation of upper bound on executing times, such as for hard real-time systems, performance estimation methods are more suitable. Performance estimation techniques are often assisted by program flow analysis techniques and micro-architectural analysis. Program flow analysis is used to analyse and represent the path constraints within a program while micro-architectural analysis is used to analyse the underlying micro-architecture. We also presented a general micro-architectural modeling framework using abstract interpretation and satisfiability checking. This framework can be used to substantially improve the accuracy of WCET analysis in the presence of many infeasible paths in a program.

Unlike performance estimation techniques which uses static analysis, profiling relies on dynamic analysis techniques. Profiling techniques do not model the underlying micro-architecture, as a result they can be light-weight. However, no guarantees about the completeness of the results (generated through profiling) can provided. Additionally, profiling techniques require test-inputs to generate the profile. Manually obtaining test-inputs that can represent the entire input-space or highlight suboptimal behaviour is non-trivial. Hence, there is a need for performance-aware test-generation techniques, that can automatically explore the program's input space and generate test-inputs that lead suboptimal behaviour. One of our works address this need and is described in the following chapter.

3 STATIC ANALYSIS DRIVEN CACHE PERFORMANCE TESTING

Real-time, embedded software are constrained by several non-functional requirements, such as timing. With the ever increasing performance gap between the processor and the main memory, the performance of memory subsystems often pose a significant bottleneck in achieving the desired performance for a real-time, embedded software. Cache memory plays a key role in reducing the performance gap between a processor and main memory. Therefore, analysing the cache behaviour of a program is critical for validating the performance of an embedded software. In this chapter, we propose a novel approach to automatically generate test inputs that expose the cache performance issues to the developer. Each such test scenario points to the specific parts of a program that exhibit anomalous cache behaviour along with a set of test inputs that lead to such undesirable cache behaviour. We build a framework that leverages the concepts of both static cache analysis and dynamic test generation to systematically compute the cache-performance stressing test inputs. Our framework computes a test-suite which does not contain any *false positives*. This means that each element in the test-suite points to a *real cache performance issue*. Moreover, our test generation framework provides an assurance of the test coverage via a well-formed coverage metric. We have implemented our entire framework using Chronos worst case execution time (WCET) analyzer and LLVM compiler infrastructure. Several experiments suggest that our test generation framework quickly converges towards generating cache-performance stressing test cases. We also show the application of our generated test-suite in design space exploration and cache performance optimization.

3.1 NEED FOR PERFORMANCE TESTING

Real-time embedded systems are required to satisfy several non-functional properties, such as timing. Therefore, performance validation marks a crucial stage before certifying such time-critical software. In the absence of appropriate performance-validation techniques, the deployed software may suffer from severe performance problems, such as missing deadlines. For example, in the context of an *anti lock braking systems* (ABS), missing a deadline may lead to serious accidents, potentially costing human lives.

Due to the inherent gap between *processor* and *memory* performances, memory subsystems may significantly affect the performance of an embedded software. To reduce such effect, a fast cache memory is often employed between a processor and main memory. In a modern embedded processor cache memories are several magnitudes faster than the main memory. Therefore, at any point in execution, the content of the cache memory significantly impacts the performance of the underlying embedded software. The content of a cache is managed at runtime and such content depends on the accessed memory block sequence. Since different inputs to the same application may follow different execution paths, the sequence of accessed memory blocks in an execution critically depends on the input provided to the application. As a consequence, the performance of caches (and hence the performance of an application) critically depends on the input provided to the underlying embedded software.

In this chapter, we propose a novel approach to automatically generate test inputs that expose performance problems due to memory subsystems. In particular, we generate test inputs to automatically detect performance stressing memory access sequences. Such *poor* memory access sequences are *undesirable*, as they may lead to critical cache performance issues, specifically *cache thrashing* at runtime. We propose a test generation framework that

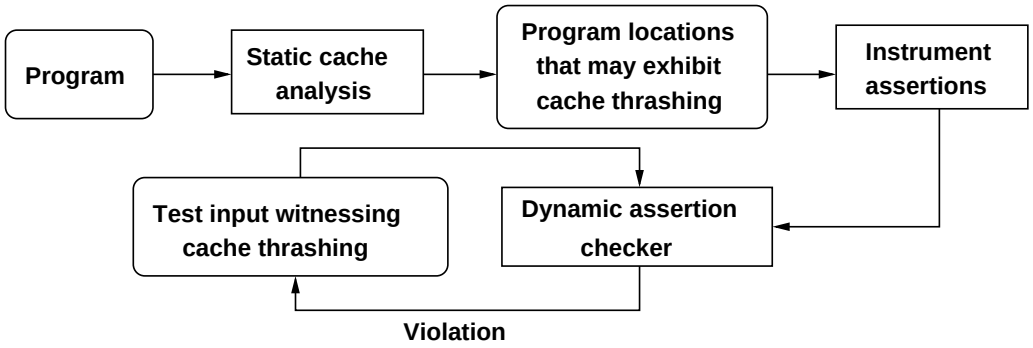
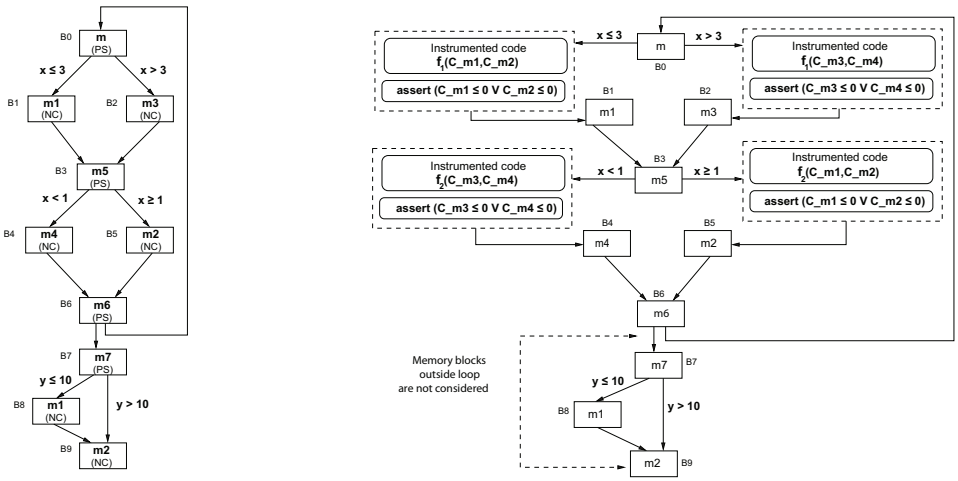


Figure 11: Test generation framework

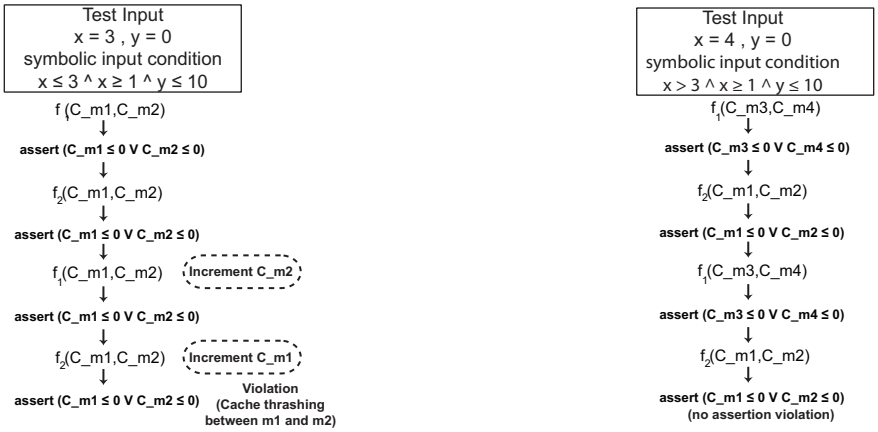
aims to report cache thrashing scenarios that exist in some program execution. Each element in our report contains a *unique* cache thrashing scenario and a symbolic formula capturing the set of inputs that expose the issue in a program execution.

However, the generation of cache-performance stressing test inputs requires solving several technical challenges. This is primarily due to the fact that cache performance issues cannot be detected solely by monitoring the program execution (unlike most of the problems in functionality testing). To overcome this problem, we employ novel strategies to instrument the original program with a set of *assertions* at appropriate locations. Such an instrumentation is *entirely automatic*. The violation of any assertion captures a *unique* cache thrashing scenario in the original program (and not in the program instrumented with assertions). Thus, such assertion violations can be reported to the developer for investigation. We first carry out static cache analysis on the program to decide the set of program points that *may* exhibit cache thrashing. Subsequently, we systematically generate assertions at such places to expose cache thrashing in the program itself. *In a broader view, therefore, we reduce the problem of testing cache performance to an equivalent functionality testing problem.* The required functionality of the software is augmented with the set of assertions introduced by us.

To check the validity of different assertions, we build a *dynamic path exploration* strategy that directs the path searching process towards the set of instrumented assertions. Each time an assertion is encountered during execution, its *validity* is checked *on-the-fly*. If an assertion is not satisfied during program execution, a cache-performance issue is recorded along with the respective input state (*i.e.* the set of inputs that leads to the violation of the assertion, cf. Figure 11). Primary objective of the path exploration strategy is to check maximum number of unique assertions, in a given amount of time. Therefore, to improve the search efficiency of path exploration, we direct the search process towards a control flow that has maximum number of unchecked assertions *control dependent* on it. Such a directed search is accomplished by consulting the control dependency graph of the instrumented program. Finally, since we dynamically explore the set of assertions, our computed test-suite does not contain any *false positives*. Precisely, any test case included in the computed test-suite captures a cache performance issue (specifically, a cache thrashing scenario) in some *feasible execution* of the software.



(a) (b)



(c)

Figure 12: Overview of test generation (a) Control flow graph showing accessed memory blocks (b) instrumented program (c) violation of assertion showing cache thrashing scenario

3.2 STATIC ANALYSIS DRIVEN CACHE PERFORMANCE TESTING: AN OVERVIEW

In this section, we shall give an outline of our test generation process that stresses the cache performance of a program. We shall walk through a simple example as shown in Figure 12. For the sake of illustration, let us assume a direct-mapped cache where memory blocks $\{m_1, m_2, m_3, m_4, m, m_5, m_6, m_7\}$ in the control flow graph are mapped to different cache sets $\{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4, \mathcal{S}_5, \mathcal{S}_6\}$ as follows: $m1 \mapsto \mathcal{S}_1$, $m2 \mapsto \mathcal{S}_1$, $m3 \mapsto \mathcal{S}_2$, $m4 \mapsto \mathcal{S}_2$, $m \mapsto \mathcal{S}_3$, $m_5 \mapsto \mathcal{S}_4$, $m_6 \mapsto \mathcal{S}_5$ and $m_7 \mapsto \mathcal{S}_6$. Therefore, $m1$ and $m2$, as well as $m3$ and $m4$ conflict in the cache.

Figure 13 presents an overview of our test generation framework. Broadly, our approach contains two separate steps: (i) static cache analysis and (ii) dynamic test generation to expose different cache thrashing scenarios in the program. The static cache analysis directs the dynamic test generation process to explore only the relevant portions of a program. Such relevant portions capture designated program points that are more likely to expose cache thrashing behaviour.

Static cache analysis is performed via abstract interpretation [15]. Memory blocks are categorized as AH (*always cache hit*), AM (*always cache miss*) and PS (*persistent* or never evicted from the cache). If a memory block cannot be categorized as AH, AM or PS, it is categorized as NC (*not classified*). As an AH/PS categorized memory block can face only *cold cache misses*, we conclude that AH/PS categorized memory blocks can never be involved in a cache thrashing scenario. Therefore, only AM or NC categorized memory blocks exhibit potential sources of *cache thrashing*. If we employ abstract interpretation based cache analysis in the example program of Figure 12(a), we observe that memory blocks m, m_5, m_6 and m_7 are categorized as PS (note that m, m_5 and m_6 do not face any cache conflict within the loop). On the contrary, memory blocks $m1$, $m2$, $m3$ and $m4$ are categorized as NC (as $m1$ conflicts with $m2$ and $m3$ conflicts with $m4$ in the cache).

The key to our test generation approach is to create an interface between *static cache analysis* and *dynamic test generation*. Such an interface is developed via systematically generating *assertions*. The set of assertions has an *one-to-one correspondence* with the set of *cache thrashing scenarios*. The violation of any assertion exposes a unique cache thrashing scenario. Therefore, in a broader perspective, our performance testing framework can be viewed as a reduction of the cache performance problem to an equivalent functionality testing problem. Figure 12(b) demonstrates the schematic of the interface. The interface mainly consists of two parts: (i) instrumented code to count cache conflicts, and (ii) set of assertions to be checked. It is worthwhile to note that the instrumented program (*i.e.* Figure 12(b)) may have a different cache behaviour compared to the original program. This is due to the presence of additional instrumented code in Figure 12(b). However, the instrumented code (*i.e.* functions f_1 and f_2) as well as the assertions (cf. Figure 12(b)) take input from memory blocks in the original program (*i.e.* memory blocks $m1$,$m2$,$m3$ and $m4$ in Figure 12(a)). Therefore, violation of any assertion captures a cache thrashing scenario in the original program shown in Figure 12(a) (and not in the instrumented program shown in Figure 12(b)).

Let us first consider the set of memory blocks $\{m1, m2\}$ in Figure 12(b). C_m1 (C_m2) captures the amount of cache conflicts generated to memory block $m1$ ($m2$). Specifically, for *least recently used* (LRU) cache replacement policy, C_m1 (C_m2) captures the number of *unique cache conflicts (i.e. number of unique memory blocks mapping to the same cache set) between two consecutive accesses of memory block* $m1$ *(*$m2$*)*. Therefore, if $C\_m1 > 0$ (recall that we assumed a direct-mapped cache) before accessing memory block $m1$, accessing $m1$ will result in a *cache miss*. The instrumented code essentially manipulates the set of variables $\{C\_m1, C\_m2, C\_m3, C\_m4\}$ through some additional code fragments. At this

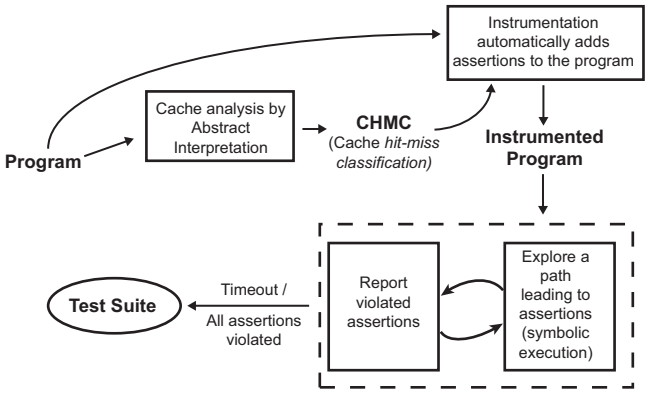


Figure 13: Overview of our test generation framework

point, without going into the details of instrumentation, we represent the instrumentation as *functions* to show the specific variables they manipulate. As shown in Figure 12(b), a function $f_1(C\_m1, C\_m2)$ only manipulates C_m1 and C_m2 (and neither C_m3 nor C_m4). In general, a cache miss does not necessarily capture a *cache thrashing scenario*. For the set of memory blocks $\{m1, m2\}$, we informally say that a cache thrashing happens when *both* $m1$ *and* $m2$ *are evicted from the cache at least once*. Therefore, the cache thrashing scenario involving memory blocks $m1$ and $m2$ is captured by the following formula:

$$\Phi_{12} \equiv C\_m1 > 0 \wedge C\_m2 > 0$$

The placed assertion checks the formula $\neg\Phi_{12} \equiv C\_m1 \leq 0 \vee C\_m2 \leq 0$ during dynamic test generation process. As a result, any violation of the assertion (formula $\neg\Phi_{12}$) captures a *cache thrashing scenario* in a *real execution*. The cache thrashing scenario involving memory blocks $\{m3, m4\}$ can be captured in a similar fashion using the formula $\Phi_{34} \equiv C\_m3 > 0 \wedge C\_m4 > 0$. Therefore, the violation of $\neg\Phi_{34}$ during the dynamic test generation process will capture a *real cache thrashing scenario* involving memory blocks $m3$ & $m4$.

Let us now investigate our dynamic test generation process. The primary goal of the dynamic test generation module is to stress the execution towards the set of instrumented assertions. The idea of our dynamic test generation has been inspired by recent advances in *satisfiability modulo theory* (SMT) and constraint-based test generation [5]. Our test generation module first executes the instrumented program with a random input, records the set of *violated assertions* (*i.e.* the set of real cache thrashing scenarios) and collects the constraints along the executed path. We assume x and y are inputs to the program. Figure 12(c) captures the execution trace for an input $x = 3, y = 0$. Due to the increment of both C_m1 and C_m2 (by the instrumented code $f_1(C\_m1, C\_m2)$ and $f_2(C\_m1, C\_m2)$, respectively), the assertion $assert(C\_m1 \leq 0 \vee C\_m2 \leq 0)$ is violated (as shown in Figure 12(c). Such an assertion violation captures the cache thrashing scenario involving memory blocks $m1$ and $m2$. To drive the execution towards other assertions, we first collect the constraints along the current execution trace. For an input $x = 3, y = 0$, such constraints can be expressed by the formula $x \leq 3 \wedge x \geq 1 \wedge y \leq 10$. To execute a different path, one of the branch conditions (*i.e.* $x \leq 3$, $x \geq 1$ or $y \leq 10$) must be negated [5]. Our test generation employs strategies to systematically negate the branches, so that the execution may lead to *maximum number of unchecked assertions*.

To check maximum number of assertions, we consult the control dependency graph (CDG) of a program. CDG captures the set of control conditions that are *necessary* to execute a certain statement. Figure 14 shows the CDG of Figure 12(b). The two assertion from the 12(b)

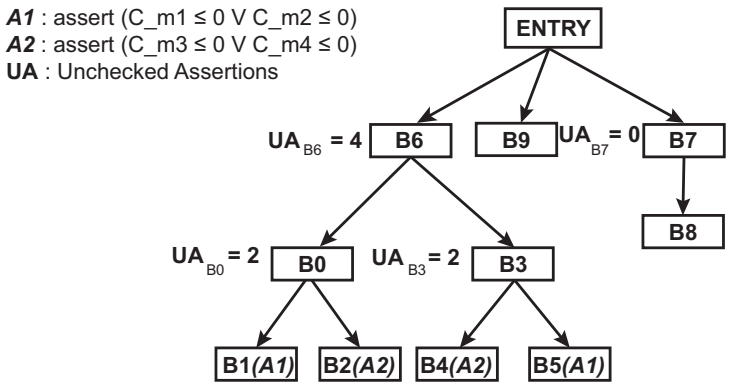


Figure 14: Control Dependence Graph, for Figure 12(a)

as shown as literals $A1$ and $A2$ in the CDG. The value against each control dependency nodes denotes the maximum number of unchecked assertion (UA) reachable from that node. In the example shown in Figure 12(b) three control conditions $x \leq 3$, $x \geq 1$ and $y \leq 10$ correspond to blocks $B0$, $B3$ and $B7$ respectively. As can be observed from Figure 14, negating the control condition at $B7$ (*i.e.* $y \leq 10$) will not lead to any unchecked assertions. Therefore, we must negate the control conditions at $B0$ (*i.e.* $x \leq 3$) or $B3$ (*i.e.* $x \geq 1$). In general, our method employs a *greedy strategy* to pick a control condition, which can lead to maximum number of unchecked assertions. Assume that branch $x \leq 3$ is chosen for negation and we obtain a test input $x = 4, y = 0$ for the symbolic condition $x > 3$. Executing the program for $x = 4, y = 0$ never violates any assertions. Note that the formula $x > 3 \wedge x < 1$ must be satisfied to execute both $f_1(C\_m3, C\_m4)$ and $f_2(C\_m3, C\_m4)$. However, the formula $x > 3 \wedge x < 1$ is clearly *unsatisfiable*. Therefore, $x > 3 \wedge x < 1$ captures an *infeasible path* in Figure 12(a) and $f_1(C\_m3, C\_m4)$ and $f_2(C\_m3, C\_m4)$ cannot appear in any execution trace together. As a result, the assertion $assert(C\_m3 \leq 0 \vee C\_m4 \leq 0)$ is always validated.

In the end, for the example shown in Figure 12, our framework finds *exactly one* cache thrashing scenario (that involves memory blocks $m1$ and $m2$) and a test input capturing the same thrashing scenario (*i.e.* $x = 3$ for a symbolic formula $x \leq 3 \wedge x \geq 1$). Our framework guarantees to cover *all the assertions* at the end of the test generation method. Note that due to the inherent limitations imposed by constraint solvers the test generation process may go on forever. However, our test generation has the *anytime* property, meaning that the test generation process can be terminated anytime if the time budget is violated. After such a premature termination, the computed test-suite exposes a subset of thrashing scenarios that exist in the program. In fact, due to our directed search via CDG, our experimental results suggest that we can find most thrashing scenarios very early in the test generation process.

SYSTEM AND APPLICATION MODEL In this work, we shall assume the traditional configuration of WCET analysis. Therefore, we consider only uninterrupted executions of a program and the computed thrashing scenarios appear solely due to the *intra-task* variant of cache conflicts. However, given a set of preemption points, our technique can be extended to capture thrashing scenarios that may appear only in the presence of preemptions. Such an extension will need to instrument the preempting tasks to compute the inter-task cache conflicts and it will require to shift the execution across tasks during test generation. Moreover, for the sake of simplicity, our framework is shown for separated instruction and data caches. To consider unified caches, the computation of cache conflicts can be combined during instrumentation (*i.e.* the computation of variables $\{C\_m1, \ldots, C\_m4\}$ in Figure 12).

3.3 TEST GENERATION METHODOLOGIES

In this section, we shall describe our test generation methodologies in detail. Broadly, our test generation methodology contains two substeps: (i) systematically generating assertions to expose cache thrashing behaviour and (ii) a *dynamic* test generation to check the validity of the generated assertions. We shall elaborate these two steps in the following sections. For the sake of simplicity, we shall describe the core methodologies for instruction caches and we shall mention the minor changes required in the instrumentation to handle data caches.

3.3.1 Generating Assertions

Code Instrumentation

Figure 15 shows the instrumented code for our example program in Figure 12. We assume that memory blocks $m1$ and $m2$ conflict in a direct-mapped cache. Therefore, after the static cache analysis, both $m1$ and $m2$ are categorized as *unclassified* (NC). Informally, the instrumented code manipulates the cache conflict faced by a particular memory block. Such an instrumentation depends on the underlying *cache replacement policy*. For the sake of illustration, we shall use *least recently used* (LRU) cache replacement policy. However, such an instrumentation can easily be changed for other cache replacement policies (*e.g.* FIFO) in a similar fashion as in the work of [18].

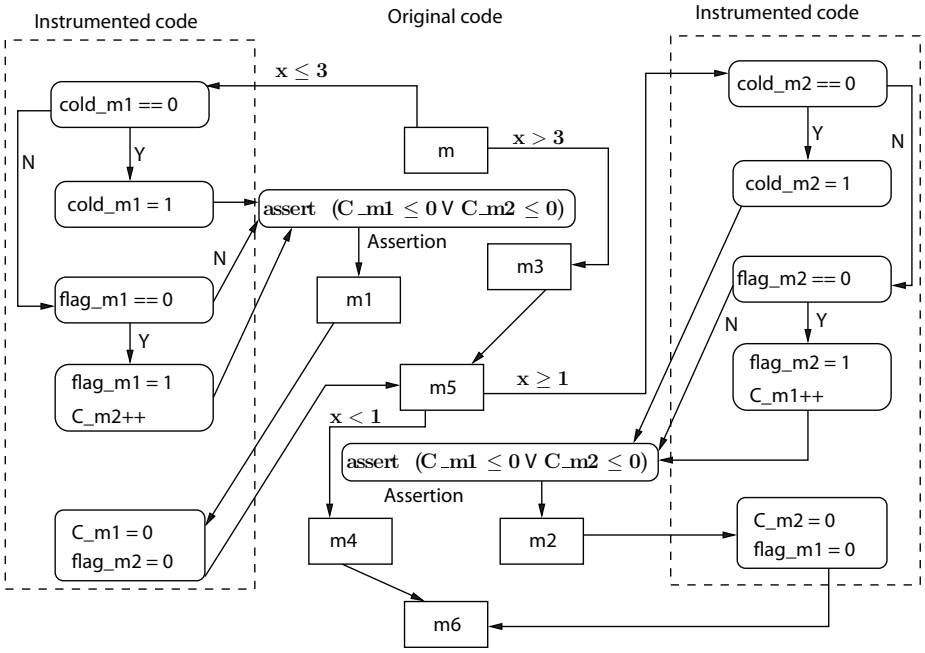


Figure 15: Instrumented code with assertions

The heart of the instrumentation shown in Figure 15 lies in manipulating the two variables C_m1 and C_m2. For LRU cache replacement policy and a particular memory block m, C_m *captures the number of unique cache conflicts between two consecutive accesses of memory block* m [1]. While counting such cache conflicts, we do not count the conflicts generated merely due to *cold misses*. Let us consider the instrumented code before memory block $m1$

1 For FIFO cache replacement policy, C_m captures the number of unique cache conflicts faced by m since it is last reloaded into the cache [18]

(as shown in Figure 15). Since memory block $m1$ creates conflicts to only memory block $m2$, such cache conflicts are captured by the increment of variable C_m2. Variable $flag\_m1$ is used to count only *unique cache conflicts* (*i.e.* the number of unique memory blocks conflicting with memory block $m1$). Besides, variable $cold\_m1$ is used to discard the *cold cache miss* for accessing memory block $m1$. The instrumented code introduced for memory block $m2$ is entirely symmetric to the one introduced for block $m1$.

Formulation of assertions

The crucial step of the instrumentation is to systematically inserting assertions to expose *cache thrashing*. Cache thrashing behaviour only happens inside program loops. Therefore, for rest of the discussion, we shall only consider memory blocks inside program loops. Moreover, without loss of generality, we shall consider memory blocks mapped to a single cache set. For set-associative caches, the process is identically applied for each cache set.

The formulation of an assertion depends on the definition of *cache thrashing*. Therefore, we first formally define the notion of *cache thrashing* used in this work.

Definition 3.3.1 Consider a $\mathcal{K}$-way set associative cache. A set of memory blocks $\mathcal{M} := \{M_1, M_2, \ldots, M_{\mathcal{K}+1}\}$ is said to have cache thrashing if and only if, for all $i \in [1, \mathcal{K}+1]$, access to M_i suffers at least one non-cold miss and all the cache conflicts for this non-cold miss are generated by the set of memory blocks $\mathcal{M} \setminus \{M_i\}$.

In the preceding definition of cache thrashing, the number of *non-cold* misses (say $\mathcal{X}$) is a tunable parameter. In our work, we assume $\mathcal{X} = 1$. However, in the following, we show that our technique can be generalized for different values of $\mathcal{X}$.

The instrumented code in Figure 15 takes the accessed memory blocks in the original program (*i.e.* the set of memory blocks $\{m1, m2, m3, m4\}$ in Figure 12(a)) as input. Therefore, it is worthwhile to note that the instrumented code manipulates cache conflicts in the original program (*i.e.* Figure 12(a)) and not the instrumented program shown in Figure 15. Since the validity of inserted assertions are based on this instrumented code, any violation of an assertion essentially captures a cache thrashing scenario in the original program (*i.e.* the program shown in Figure 12(a)).

To describe the generation of assertions, we shall begin with a few notations and definitions. Let us assume $\mathcal{M}_l = \{M_1, M_2, \ldots, M_N\}$ is the set of memory blocks accessed inside some program loop. We define a *thrashing set* as a subset of $\mathcal{M}_l$ that may be potentially involved in cache thrashing. Formally, a thrashing set $\mathcal{TS}_l$ is defined as follows

$$\mathcal{TS}_l = \{m \mid m \in \mathcal{M}_l \wedge CHMC(m) \neq PS \wedge CHMC(m) \neq AH\} \tag{27}$$

In Equation 27, *CHMC* captures the *cache hit-miss classification* obtained via static cache analysis [15]. Note that *AH* (all-hit) and *PS* (persistent) categorized memory blocks can never be evicted from the cache (due to the inherent guarantee provided by static analysis). Therefore, we do not include such memory blocks as the potential cause of cache thrashing.

From a *thrashing set*, we define a number of *thrashing scenarios*. Informally, a cache thrashing scenario contains *just enough* memory blocks from a thrashing set to create a potential cache thrashing. If we assume that the associativity of the cache is $\mathcal{K}$, the *minimum* number of memory blocks to create a cache thrashing is $\mathcal{K}+1$. Therefore, a *thrashing scenario* for a thrashing set $\mathcal{TS}_l$ is defined as any $\mathcal{K}+1$ combination of the thrashing set $\mathcal{TS}_l$. The set of all cache thrashing scenarios Ω_l can be defined as follows.

$$\Omega_l = \{\, S \subseteq \mathcal{TS}_l \mid |S| = \mathcal{K}+1\} \tag{28}$$

Note that a thrashing set $\mathcal{TS}_l$ has a total of $\binom{|\mathcal{TS}_l|}{\mathcal{K}+1}$ different cache thrashing scenarios.

Finally, we generate exactly one assertion for each cache thrashing scenario. Let us assume one such cache thrashing scenario $\Theta \in \Omega_l$ and its respective assertion $\mathcal{A}_\Theta$. Informally, the assertion $\mathcal{A}_\Theta$ captures the property that thrashing scenario Θ *never happens in any program execution*. As a result, any violation of the assertion $\mathcal{A}_\Theta$ during dynamic test generation captures a realization of the thrashing scenario Θ. Formally, thrashing scenario Θ is captured by the following property.

$$\Phi_\Theta \equiv \bigwedge_{m \in \Theta} (C_m > \mathcal{K}) \tag{29}$$

In Equation 29, C_m captures the amount of unique cache conflicts faced by two consecutive accesses of memory block m. Since the assertion checks the negation of thrashing scenario, it can be formalized as follows.

$$\mathcal{A}_\Theta \equiv assert(\neg \Phi_\Theta) \tag{30}$$

The assertion $\mathcal{A}_\Theta$ is placed before each memory block involved in the thrashing set Θ. For example, in Fig. 15, the set $\{m1, m2\}$ captures a thrashing scenario and the assertion $assert(C\_m1 \leq 0 \vee C\_m2 \leq 0)$ was placed before accessing memory blocks $m1$ and $m2$.

The purpose of the preceding assertion (Eq. 30) is to validate that *at least one of the memory block from the thrashing set is never evicted from the cache*. Therefore, if all of the memory blocks in a thrashing set are evicted *at least once*, an assertion violation will be triggered and a cache performance issue will be reported. The assertion $\mathcal{A}_\Theta$ is checked dynamically before accessing each memory block involved in the thrashing scenario Θ.

It is worthwhile to mention that our formalization to capture cache thrashing (*i.e.* Definition 3.3.1 and Equation 29) is independent of cache replacement policy. Therefore, such a formalization can be applied to a wide variety of cache architectures. However, the instrumentation which is carried out to transform the program code depends on cache replacement policy. For instance, in the Least Recently Used replacement policy, a cache hit causes the cache state to be updated such that the age of the most recently accessed memory block is 0 however in the First In First Out cache replacement the cache state is not changed after a cache hit. These difference are reflected in the computation of the variable C_m (cf. Section 3.3.1). Finally, we can also generalize the notion of reporting a cache thrashing scenario at runtime. Specifically, a cache thrashing scenario can be reported for $\mathcal{X}$ number of violations of an assertion (and thereby $\mathcal{X}$ number of evictions for each memory block in the respective thrashing set) instead of only one violation. Such a generalization also corresponds to the reconfiguration of the number of non-cold misses, as described in Definition 3.3.1.

Handling data caches

For data caches, the memory block classification is obtained using the *scope-aware persistent* (SCP) analysis [23]. SCP analysis can be used to classify data memory blocks as persistent or non-persistent. Unlike the instruction cache analysis, determining the set of data memory blocks accessed at a program point, can be challenging. Existing address analysis techniques such as the one used in [23] can be used to obtain the set of memory blocks, which may be accessed at a given program point. Once the SCP analysis has been performed on the set of memory blocks generated by address analysis, the assertions can be generated as described in preceding paragraphs.

The basic structure for instrumentation is similar to what was described for instruction caches. However, unlike an instruction access, a data access may correspond to multiple memory blocks. For example, in Figure 16, access to Array_X might result in fetching of memory block $m1$ or $m2$, depending upon the loop iteration. Likewise, access to Array_Y might result in fetching of memory block $m3$ or $m4$.

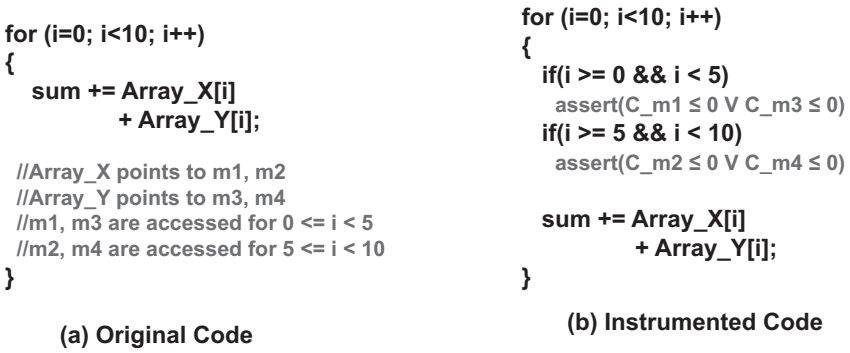


Figure 16: Instrumentation scenarios for data caches

Assume that the address analysis has reported that the memory blocks $m1$ and $m3$ are accessed for loop iterations i $\in [0..4]$ and memory blocks $m2$ and $m4$ are accessed for loop iterations i $\in [5..9]$. Also assume that the only sets of memory blocks which conflict in the cache are $\{m1,m3\}$ and $\{m2,m4\}$. Under these assumptions, memory block $m1$ and $m3$ can participate in a thrashing scenario, only during loop iterations $[0..4]$. Therefore, the instrumented code for $m1$ and $m3$ needs to be proceeded by conditional checks on the iteration number ($i >= 0$ && $i < 5$). Conditional checks for memory block $m2$ and $m4$ can be placed in a similar fashion. Figure 16(b) shows the instrumented code for the example program shown in Figure 16(a).

3.3.2 Dynamic Test Generation

Dynamic test generation tries to find violations of the instrumented assertions (cf. Section 3.3.1). Our dynamic test generation process is inspired by recent advances in *constraint solving* and *concolic* testing [5]. As an output of the dynamic test generation process, we obtain a pair $\langle \Theta_i, \Psi_i \rangle$ for each cache thrashing scenario Θ_i. In the output pair, Ψ_i captures a symbolic formula on the input variables, such that any input satisfying the formula leads to the cache thrashing scenario Θ_i. In our example (cf. Figure 12), one such output would be $\langle \{m1, m2\}, x \leq 3 \wedge x \geq 1 \rangle$. This implies that any input value of x $\in$ [1,3], would lead to a thrashing scenario, involving memory blocks m_1 and m_2.

Algorithm 1: The primary goal of Algorithm 1 is to check the validity of instrumented assertions (cf. Section 3.3.1). It takes the instrumented program $\mathcal{P}_\mathcal{A}$ and the set of instrumented assertions $\mathcal{A}$ as inputs and generates a set of test cases $\mathcal{T}$. Each element in $\mathcal{T}$ realizes a unique cache thrashing scenario. To begin with, Algorithm 1 executes the instrumented program $\mathcal{P}_\mathcal{A}$ with a random input $\mathcal{I}$ and collects the execution trace $\mathbb{X}$. The exploration of different assertions is performed by systematically manipulating the *path condition* of this execution trace. Formally, *path condition* is defined as follows.

Definition 3.3.2 For a particular execution trace $\mathbb{X}$, path condition is a quantifier free first order logic formula that captures exactly the set of inputs which drives the program execution through the execution trace $\mathbb{X}$.

Algorithm 1 Dynamic exploration of instrumented assertions

Input:
$\mathcal{P}_{\mathcal{A}}$: instrumented program with assertions
$\mathcal{A}$: set of instrumented assertions
Output:
$\mathcal{T}$: a set of test cases, each of which realizes a unique cache thrashing scenario
$AllPathConditions = unchecked = \mathcal{T} =$ empty
/*build the control dependency graph (CDG) of $\mathcal{P}_{\mathcal{A}}$ */
$CDG_{\mathcal{P}_{\mathcal{A}}} \leftarrow$ `BuildCDG`($\mathcal{P}_{\mathcal{A}}$)
select a random input $\mathcal{I}$
`ExecuteAndReport`($\mathcal{P}_{\mathcal{A}}$,$\mathcal{A}$, $\mathcal{I}$,$CDG_{\mathcal{P}_{\mathcal{A}}}$)
while $unchecked \neq$ empty $\wedge$ $\mathcal{A} \neq$ NULL **do**
    /*pick a partial path with maximum number
    of reachable and non-violated assertions */
    select $\langle \varphi, \mathcal{F}_{\varphi} \rangle \in unchecked$ with maximum $\mathcal{F}_{\varphi}$
    $unchecked := unchecked \setminus \{\langle \varphi, \mathcal{F}_{\varphi} \rangle\}$
    let $\varphi \leftarrow \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{r-1} \wedge \psi_r$
    /* execute an unexplored path */
    if φ is satisfiable **then**
        $\tau_{\varphi} \leftarrow$ some concrete inputs satisfying φ
        `ExecuteAndReport`($\mathcal{P}_{\mathcal{A}}$, $\mathcal{A}$, τ_{φ},$CDG_{\mathcal{P}_{\mathcal{A}}}$)
    end if
end while
procedure EXECUTEANDREPORT($\mathcal{P}_{\mathcal{A}}$,$\mathcal{A}$,$\tau$,$CDG_{\mathcal{P}_{\mathcal{A}}}$)
    execute $\mathcal{P}_{\mathcal{A}}$ on input τ
    let $\mathbb{X}$ be the execution trace on input τ
    let $\varphi \equiv \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_k$ be the path condition
    let $\{\mathcal{A}_{\theta_1}, \mathcal{A}_{\theta_2}, \ldots, \mathcal{A}_{\theta_r}\}$ be the set of *violated*
    assertions, on input τ
    $\mathcal{A} = \mathcal{A} \setminus \{\mathcal{A}_{\theta_1}, \mathcal{A}_{\theta_2}, \ldots, \mathcal{A}_{\theta_r}\}$
    let π be the *shortest* path-prefix along the execution
    trace $\mathbb{X}$ that captures at least one violation of each
    assertion in the set $\{\mathcal{A}_{\theta_1}, \mathcal{A}_{\theta_2}, \ldots, \mathcal{A}_{\theta_r}\}$
    let $\hat{\varphi}$ captures the partial path condition corresponding
    to the path-prefix π
    /* augment the test suite with witnesses for cache
    thrashing scenarios */
    $\mathcal{T} \bigcup = \{\langle \theta_1, \hat{\varphi} \rangle, \langle \theta_2, \hat{\varphi} \rangle, \ldots, \langle \theta_r, \hat{\varphi} \rangle\}$
    /*build all partial path conditions */
    for $i \leftarrow 1, k$ **do**
        let $\varphi_i \leftarrow \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$
        if $\varphi_i \notin AllPathConditions$ **then**
            $AllPathConditions \bigcup = \varphi_i$
            let b_{end} be the control dependency edge in
            $CDG_{\mathcal{P}_{\mathcal{A}}}$ w.r.t. the branch condition $\neg\psi_i$
            /* compute the number of non-violated
            assertions ($\in \mathcal{A}$) reachable from b_{end} */
            $\mathcal{F}_{\varphi_i} :=$ `GuidanceFunction`(b_{end})
            if $\mathcal{F}_{\varphi_i} \neq 0$ **then**
                $unchecked \bigcup = \{\langle \varphi_i, \mathcal{F}_{\varphi_i} \rangle\}$
            end if
        end if
    end for
end procedure

For example, in Figure 12(c), the symbolic formula $x \leq 3 \wedge x \geq 1 \wedge y \leq 10$ captures the *path condition* for the execution trace on input values $x = 3$ and $y = 0$.

The variable *unchecked* represents the set of unexplored partial path conditions in the instrumented program $\mathcal{P}_{\mathcal{A}}$. Each partial path condition φ_i is associated with a metric $\mathcal{F}_{\varphi_i}$. This metric measures the maximum number of non-violated assertions reachable from φ_i. We employ a *greedy strategy* based on the value of $\mathcal{F}_{\varphi_i}$ to continue exploration. More precisely, we generate a test input τ_θ from an unexplored, partial path condition φ_i that has the maximum value for $\mathcal{F}_{\varphi_i}$. Subsequently, we invoke the procedure *ExecuteAndReport* with input τ_θ.

Procedure ExecuteAndReport executes the instrumented program $\mathcal{P}_{\mathcal{A}}$ for an input τ to obtain the execution trace $\mathbb{X}$. For a particular execution, assume that $\varphi \equiv \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1} \wedge \psi_k$ captures the path condition for the execution trace $\mathbb{X}$. Further assume that $\{\mathcal{A}_{\theta_1}, \mathcal{A}_{\theta_2}, \ldots, \mathcal{A}_{\theta_r}\}$ is the set of violated assertions and π is the shortest path-prefix in the $\mathbb{X}$ which captures *at least one violation* of each assertion in the set $\{\mathcal{A}_{\theta_1}, \mathcal{A}_{\theta_2}, \ldots, \mathcal{A}_{\theta_r}\}$. Therefore, an assertion, if violated beyond path-prefix π, must belong to the set $\{\mathcal{A}_{\theta_1}, \mathcal{A}_{\theta_2}, \ldots, \mathcal{A}_{\theta_r}\}$. If φ is the respective path condition, say $\hat{\varphi}$ be the prefix of the path condition corresponding to path-prefix π. Note that $\varphi \vDash \hat{\varphi}$, however, $\hat{\varphi}$ is a compact yet *lossless* formula to manifest the set of thrashing scenarios (*i.e.* the set of thrashing scenarios exposed by violations of the set of assertions $\{\mathcal{A}_{\theta_1}, \mathcal{A}_{\theta_2}, \ldots, \mathcal{A}_{\theta_r}\}$). Therefore, for each thrashing scenario θ_i, we construct a test pair $\langle \hat{\varphi}, \theta_i \rangle$ and add such test pairs to the existing test suite $\mathcal{T}$. To avoid any redundant computation, we manipulate $\hat{\varphi}$ *on-the-fly* during the execution.

To continue exploration, we must deviate from the present path. Such a deviation is performed by negating a branch conditions along the execution trace $\mathbb{X}$. Assume that we pick a partial path condition $\varphi_i \equiv \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$. Let us also assume that b_{end} is the control dependency edge in the CDG (of the instrumented program) that captures the negated branch condition $\neg\psi_i$. We rank each partial path condition φ_i with a metric $\mathcal{F}_{\varphi_i}$ and add it to the set of *unchecked* partial path conditions. $\mathcal{F}_{\varphi_i}$ captures the *maximum* number of assertions in $\mathcal{A}$ that are reachable, if the program is executed with a test input satisfying φ_i.

GuidanceFunction captures the computation of $\mathcal{F}_{\varphi_i}$. Formally, $\mathcal{F}_{\varphi_i}$ is defined as follows.

$$\mathcal{F}_{\varphi_i} = |\{\mathcal{A}_\theta \in \mathcal{A} \mid b_{end} \rightsquigarrow \mathcal{A}_\theta\}| \tag{31}$$

Where $b_{end} \rightsquigarrow \mathcal{A}_\theta$ captures that the assertion $\mathcal{A}_\theta$ is reachable from the control dependency edge b_{end} (*i.e.* the control dependency edge corresponding to the negated branch condition ψ_i). Therefore, $\mathcal{F}_{\varphi_i}$ accounts for all the assertions in $\mathcal{A}$ that are reachable from b_{end}. It is worthwhile to note that the definition of $\mathcal{F}_{\varphi_i}$ (in Equation 31) can be changed very easily depending on the *criticality* of different assertions. In Equation 31, we have only considered the reachability of assertions, giving each assertion equal priority. However, the definition $\mathcal{F}_{\varphi_i}$ can be easily changed to incorporate other priorities (*e.g.* assertions in an innermost loop can be given higher priorities than assertions in an outermost loop).

TERMINATION Algorithm 1 terminates as soon as we obtain a witness (*i.e.* test case) for each cache thrashing scenario (captured by the condition $\mathcal{A} \neq \text{NULL}$ in the outermost loop of Algorithm 1). However, some thrashing scenarios might not be manifested due to the presence of *infeasible paths* in a program (*e.g.* the assertion $assert(C\_m3 \leq 0 \vee C\_m4 \leq 0)$ in Figure 12(c)). In such cases, the test generation process may go on forever in the presence of *unbounded* (*e.g.* input-dependent) loop iterations and due to the inherent *incompleteness* of any constraint solver. However, one appealing nature of our test generation process is that it can be terminated *anytime*. The resulting test-suite might be *incomplete*, but it can still be used for investigating cache performance issues in the program. Our experiments suggest that we can find most of the cache performance stressing test inputs in very early

phase of Algorithm 1. This is primarily due to the directed search strategy along the control dependency chain (via the CDG) to reach the set of instrumented assertions.

3.3.3 Salient Features of Generated Test Suites

Our generated suite has several important properties. In the following description, we shall formally capture the properties of the generated test suite.

Property 3.3.1 At any point in time during the execution of Algorithm 1, for any cache thrashing scenario Θ_i, if no path witnessing Θ_i has been explored - no test case containing Θ_i appears in the test-suite $\mathcal{T}$ computed so far. Otherwise, the entry $\langle \Theta_i, \Psi_i \rangle$ appears in the test-suite $\mathcal{T}$ where Ψ_i captures the set of all inputs which witness Θ_i, and whose paths have been explored already by Algorithm 1.

Property 3.3.2 If Algorithm 1 terminates, we can guarantee to find *a*ll feasible cache thrashing scenarios in any uninterrupted execution, that is, all cache thrashing scenarios witnessed by at least one program input. Each such feasible cache thrashing scenario will appear as an entry $\langle \Theta_i, \Psi_i \rangle$ in the generated test suite $\mathcal{T}$ reported at the end of Algorithm 1. Any solution for the formula Ψ_i is a test input witnessing cache thrashing scenario Θ_i.

3.4 EVALUATION

3.4.1 Experimental Set-up

Figure 17 shows an outline of our implementation framework. To generate cache hit-miss classifications (CHMC), we use the abstract interpretation (AI) based cache analyses (using [15] for instruction caches and using [23] for data caches) implemented in Chronos [46]. Outcomes of AI-based cache analyses are used by the instrumentation engine to compute thrashing sets and to insert assertions at appropriate program points (as explained in Section 3.3.1). This instrumented program is passed to the dynamic test generation process.

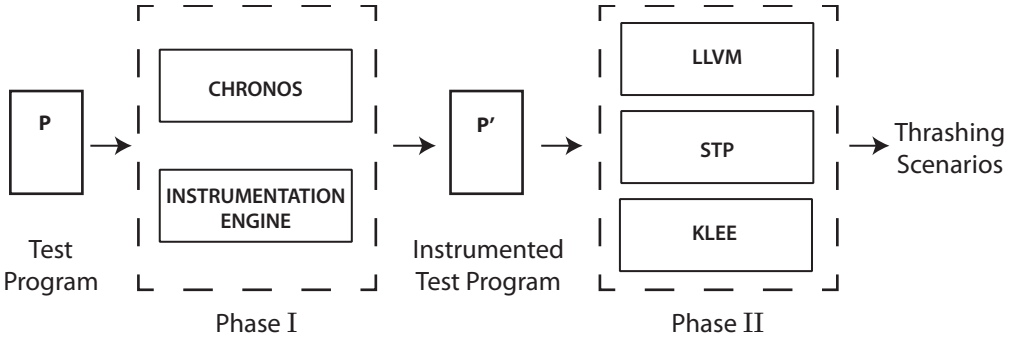
Dynamic test generation process is implemented on top of LLVM compiler infrastructure [56]. The instrumented program is compiled into LLVM bitcode format and its control flow graph (CFG) is extracted from the LLVM bitcode. We also implement a module inside LLVM to compute the control dependency graph (CDG) of a given program. This CDG is used to guide our test generation, as explained in Algorithm 1. To generate path conditions for different execution traces, we use KLEE symbolic execution engine [57]. To solve and manipulate path conditions along an execution trace, we use the STP constraint solver [58]. We have performed all the experiments on a machine having an Intel Core-i5 processor, with 4 GB RAM and running Ubuntu 9.04 OS.

SUBJECT PROGRAMS Table 4 shows the subject programs, used in our experiments. *Nsichneu*[47] is an automatically generated code, which simulates an extended Petri Net. It was taken from the Mälardalen WCET benchmarks suite. It has a code size much larger than other programs used in our experiments. Also it contains a large amount of if-statements. *Papabench*[48] is an Unmanned Aerial Vehicle (UAV) control application. In our experiments, we used the auto-navigation utility from *papabench*. The auto-navigation utility contains a lot of input dependent paths, therefore it can potentially show different thrashing scenarios for different symbolic input formulas. *Jetbench*[49] is a real-time, Jet engine performance calculator. It uses Jet engine parameters and thermodynamic equations from the NASA's

EngineSim program to perform real-time thermodynamic calculations. We use a single-threaded version of *Jetbench* for our experiments.

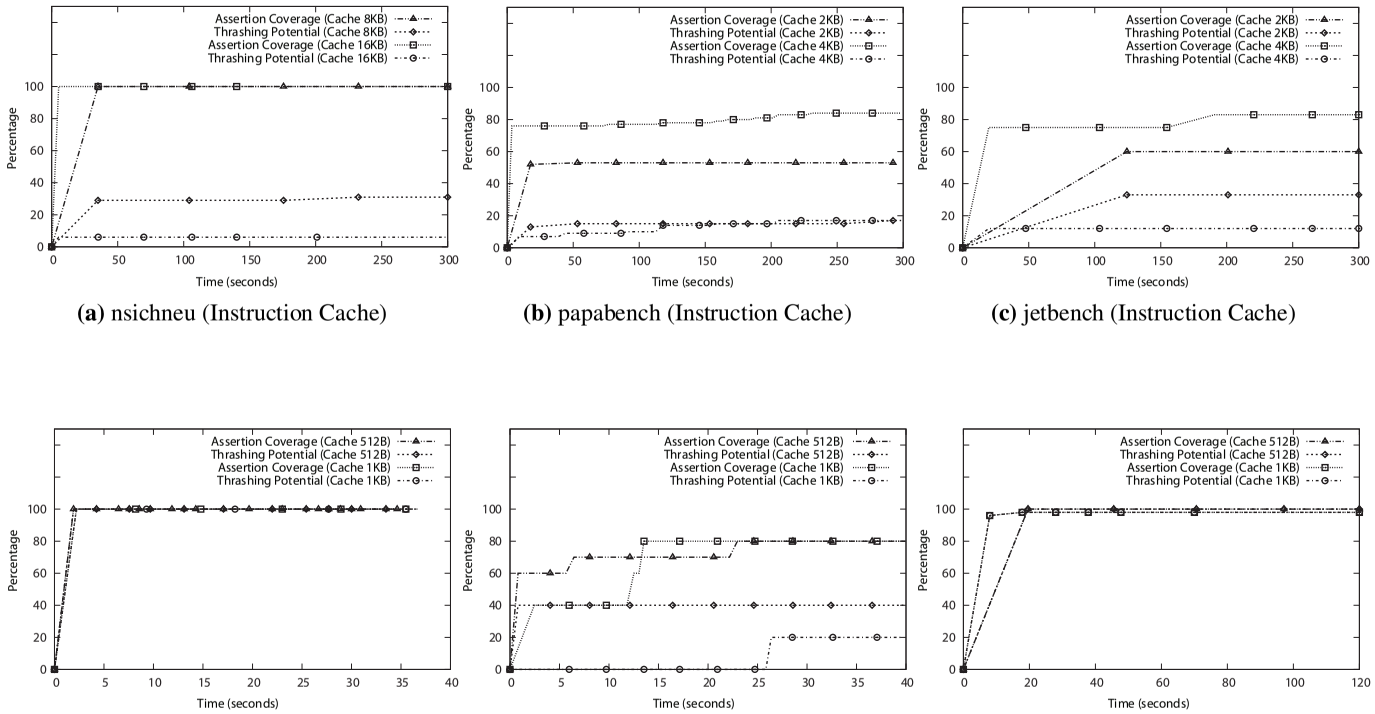
Table 4: Programs used for cache performance study

S. No	Test Program	Program Description	Lines of Code
1	Nsichneu [47]	Automatically generated code. Simulates extended Petri Net Contains large number of if-else statements	4253
2	Papabench [48]	Part of a Unmanned Aerial Vehicle controller application. Only auto-navigation utility is used	1097
3	Jetbench [49]	Real-time, Jet-engine performance calculator. User parameter and thermodynamic equations from NASA's EngineSim program	770



Tools Used:-
Chronos [46]: Static analysis based timing analysis tool
LLVM [56]: Compiler infrastructure
STP [58]: Constraint solver
KLEE[57]: Symbolic execution engine

Figure 17: Key phases in the framework



(a) nsichneu (Instruction Cache) **(b)** papabench (Instruction Cache) **(c)** jetbench (Instruction Cache)

(d) nsichneu (Data Cache) **(e)** papabench (Data Cache) **(f)** jetbench (Data Cache)

Figure 18: Assertion Coverage and Thrashing Potential for different cache configurations

3.4.2 Experimental Results

In following paragraphs, we shall describe some of the experiments which were performed to measure the efficacy of our framework. Also we shall discuss the applications of our framework for answering some of the issues related to design space exploration and performance optimization.

EFFICACY OF OUR FRAMEWORK, IN EXPOSING CACHE PERFORMANCE ISSUES We performed experiments with the three real-time programs listed in Table 4. The results of which are discussed subsequently. But first we shall describe a few metrics which are used to present the experimental results.

Assertion Coverage : Our framework aims to find all thrashing scenarios due to *intra-task cache conflicts*. However, our test generation framework may not terminate (in general, this problem is undecidable [5]). Therefore, we define a metric named *Assertion coverage* which measures the percentage of unique assertion checked, within a given amount of time.

$$\text{Assertion Coverage} = \frac{\text{unique assertion checked}}{\text{unique assertion instrumented}} \times 100$$

A 100% assertion coverage implies that all unique assertions have been checked at least once.

Thrashing Potential : It is not necessary that all the checked assertion will be violated. However, the number of unique assertions violated, tells us about the potential for cache thrashing, for a program, on a given cache-configuration. Therefore, we define *Thrashing Potential* as follows

$$\text{Thrashing Potential} = \frac{\text{unique assertion violated}}{\text{unique assertion instrumented}} \times 100$$

Through our experiments we investigated the assertion coverage and the thrashing potential for all the program listed in Table 4, for various cache-configurations. Some of the results from our experiments are shown in Figure 18. The plots in Figure 18 show the assertion coverage (and thrashing potential) on the y-axis and the exploration time on the x-axis. Since our framework looks for all possible thrashing scenarios due to intra-task cache conflicts, it is possible that the test generation will not terminate (refer to section 3.3.2). Therefore, we tested the subject programs, with an exploration budget of 5 minutes. We performed the experiments for instruction caches as well as data caches. Overall, we observed an assertion coverage ranging from 53% to 100% for different experimental set-ups (within a exploration budget of 5 minutes). Essentially, programs which had lesser number of input dependent paths (such as *nsichneu*) were explored much faster than program which had more number of input dependent paths (such as *papabench*). We also observed that for most of the experiments with instruction caches, only a small fraction of instrumented assertion were actually violated.

The figures presented in the first row (Fig. 18 (a), (b) and (c)) show the results, for instruction caches. On one hand, our chosen cache sizes are sufficient to avoid *capacity misses*. On the other hand, cache sizes are also small enough to generate *conflict misses*. Since *nsichneu* has a large code within a loop, we choose a relatively bigger cache for *nsichneu*, compared to the other two subject programs. Figure 18(a) shows the percentage coverage for *nsichneu*, on a 2-way, set-associative, LRU, instruction cache. The results reported here are for cache configuration of 8 KB and 16 KB. For both the configuration, the framework achieved a 100% assertion coverage, in less than 5 minutes. The thrashing potential for *nsichneu*, was observed to be less than 31% for both the experiments. Note that since the framework achieved a 100% assertion coverage for *nsichneu*, therefore the recorded thrashing potential is accurate. We performed experiments with *papabench* and *Jetbench* on a 2 KB and 4 KB for 2-way, set-associative, LRU cache. Neither of these experiments, resulted in a

100% assertion coverage, within the exploration budget of 5 minutes. However, this doesn't imply that the greedy exploration strategy is inefficient. This observation is supported by the fact that most of the explored assertions in Figure 18 were discovered early in the exploration. Additionally, some of the instrumented assertions may be present along infeasible paths (such as $x = 0 \wedge x = 1$), therefore they might not be checked throughout the exploration.

Figure 18 (d), (e) and (f) show the analysis results for data caches. For all the experimental results reported in this paragraph we used a direct-mapped, data caches of size of 1KB and 512B. We used small caches for this set of experiments, so as to create sufficient number of thrashing scenarios. For *nsichneu* and *Jetbench* we observed an assertion coverage of almost 100%. In fact, for *nsichneu* only one cache thrashing scenario was reported for both the cache configurations, which was covered (and violated) during exploration. Also, for *Jetbench* (see Figure 18(f)) most of the checked assertions were violated during exploration. However, for *papabench* (see Figure 18(e)), we observed an assertion coverage of 80% and a thrashing potential of less than 40%, for both the cache-configurations.

3.5 APPLICATIONS IN DESIGN SPACE EXPLORATION

The process of embedded system design can be quite challenging due to sheer size of the design space that needs to be explored. While choosing a design for an embedded application, the designer has to consider various constraints such as timing and energy consumption. For instance, while choosing a cache-configuration, a designer can choose from a large, highly-associative cache or smaller, less-associative cache. A large, highly-associative cache might have lesser number of cache-thrashing scenarios however it will consume more power and possibly slower than the smaller cache. Therefore, determining the ideal cache size for a given application might be tricky. Our framework can provide a suitable way to choose the appropriate cache configuration for an application.

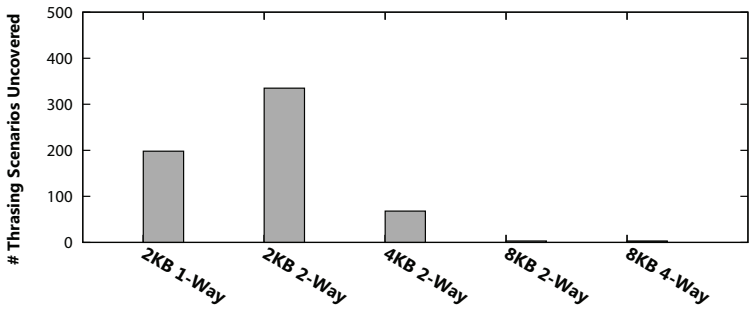


Figure 19: Number of cache thrashing scenarios discovered for papabench for various cache configurations

Essentially, our framework can be used to compare the number of thrashing scenarios for different cache configurations, for a given application. For example Figure 19 shows the number of thrashing scenarios discovered for different cache configurations, for *papabench*. It is worthwhile to note that our framework pinpoints the real thrashing scenarios, witnessed by a feasible execution. Existing techniques, which are purely based on static analysis (*e.g.* [15, 23]) may include *false thrashing scenarios* that never appear in any execution (cf. Figure 18). As a result, one can choose a more appropriate cache configuration using our framework, compared to the techniques based purely on static analysis. There exists a number of works [59] which can be used to determine the appropriate cache for a given system requirement.

In Figure 19, it might be interesting to know that a 2KB, 1-way (direct-mapped) cache has lesser number of cache thrashing scenarios then a 2KB, 2-way set-associative cache. Also, the experiments suggest that the number of cache thrashing scenarios for a 8KB, 2-way set-associative cache and a 8KB, 4-way set-associative cache are the same. So for this program, a 8KB, 2-way set-associative cache will be sufficient, to avoid cache-thrashing.

3.6 APPLICATIONS IN PERFORMANCE OPTIMIZATION

In this section, we shall discuss the application of our framework for input sensitive optimization, specifically for cache locking. The main intuition is explained via Figure 20(a). Assume a direct-mapped cache and memory blocks $m1$, $m2$, $m3$ and $m4$ all map to the same cache set. Clearly, this would result in a cache-thrashing scenario (for thrashing sets $\{m1, m2\}$ and $\{m3, m4\}$) and our test generation framework computes the following test cases: $\langle\{m1, m2\}, z \leq 5\rangle$ and $\langle\{m3, m4\}, z > 5\rangle$. In a way, therefore, our dynamic test generation framework can also be viewed as partitioning the input domain, where all inputs constituting a partition realizes the same set of cache thrashing scenarios. In our example, there are two such partitions - Δ_1 and Δ_2 (cf. Fig.20(b)).

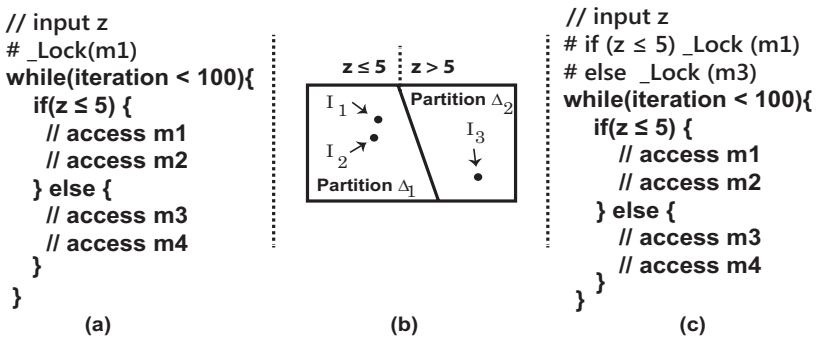


Figure 20: Illustration of conditional cache locking (a) Program with unconditional cache locking (lock instructions are preceded by #) (b) Input partitions (c) Conditional cache locking

Assume that we want to selectively lock memory blocks so that such memory blocks are never evicted from the cache. Traditional cache locking techniques, such as [6] can be used for such purposes. The work in [6] requires a memory trace (sequence of memory blocks) to determine the set of memory blocks that should be locked in the cache. However, a program might have different memory traces for different sets of inputs. If we use a memory trace generated for an input $I_1 \in \Delta_1$, either $m1$ or $m2$ will be locked in the cache (as shown in Fig 20(a)). However, it can be observed that when the program is executed for any input $I_2 \in \Delta_2$, locking $m1$ or $m2$ (as shown in 20(a)) will not improve the cache performance. This is due to the fact that $m3$ and $m4$ will encounter *cache thrashing*.

Based on the discussion in the preceding paragraph, we argue the potential of performance optimization (*e.g.* cache locking) techniques that is sensitive to inputs. In particular, for cache locking optimization in Figure 20, we could lock $m1$ (or $m2$) for all inputs satisfying $z \leq 5$ and lock $m3$ (or $m4$) for all inputs satisfying $z > 5$. Such a conditional cache locking (as shown in Figure 20(c)), will improve the program performance for both the input partitions Δ_1 and Δ_2 (cf. Figure 20(b)).

To validate our argument, we have studied the feasibility of conditional cache locking technique on the subject program `nsichneu`. For baseline cache locking optimization, we use [6], that locks a set of memory blocks from a given memory trace. We conduct several experiments for two arbitrary inputs I_1 and I_2; where I_1 is used to generate a memory trace

based on which we decide which memory blocks to lock in the cache, using the technique of [6], and I_2 is used to run `nsichneu` after the cache locking optimization is performed for the memory trace on input I_1. We have made the following crucial observations.

- If I_1 and I_2 belong to the same input partition produced by our framework, the performance improvement from cache locking observed in nsichneu is significantly greater than the situation where I_1 and I_2 belong to different input partitions. These results seem to motivate the use of conditional locking instructions.
- For the situation where I_1 and I_2 belong to the same partition, we also observed the performance improvement from locking varies across input partitions. On average, we observed a variation from $\sim$ 10% to 20% in performance improvement across different input partitions in *nsichneu*. Note that inputs from different partitions have different memory traces and so, they lead to different set of locked memory blocks using [6].

The preceding observations motivate the need for conditional cache locking, which can be studied at length in the future. Specifically, our observations conclude that memory blocks should be locked differently across different input partitions computed by our framework.

3.7 COMPARISON WITH EXISTING TECHNIQUES

Over the past two decades, a significant research effort has been put forward for the performance validation of embedded software. Such efforts include abstract interpretation (AI) based method, such as [15], which was proposed to analyze the cache behaviour of a program. The work of [18] improves the precision of such AI-based cache analyses via a gradual and controlled use of model checking. These works [15, 18] analyse the cache behaviour of a program irrespective of its inputs. On the contrary, our primary goal is to build a connection between the set of inputs and anomalous cache behaviours (*e.g.* cache thrashing). Our test generation methodology is inspired by the recent advances in constraint solving and *concolic* testing [5, 55]. These works aim to detect software functionality bugs. In contrast, we aim to detect software performance problems due to memory subsystems.

Different techniques used for *program profiling* [60, 61] also aim to find performance problems in a program. Such profiling techniques work on full or compressed execution traces to derive useful information about program performance. It is assumed that the relevant inputs for obtaining an execution trace are known *a priori*. Our approach is complementary to these profiling techniques, as our aim is to systematically find test inputs that lead to poor cache performance. Once such test inputs are found, they can be fed back to a traditional profiler for further analysis.

Recent advances in profiling [9, 10] have extended the traditional profiling technique to compute a *performance trend* of a program. Such a performance trend is captured by an approximate cost function. The cost function relates program inputs with the overall cost of the program. However, such cost functions are approximations and they do not necessarily capture the actual cost. Besides, these works do not introduce any notion of test coverage. On the contrary, any cache thrashing scenario reported by our framework is indeed a cache thrashing scenario, witnessed by a concrete input. Besides, our framework also reports the coverage of cache thrashing scenarios via the set of dynamically checked assertions.

The work proposed in [53] automatically finds test inputs for the worst-case computational complexity. Our work differs from [53] on several aspects: first, our notion of performance is based on the execution time rather than computational complexity. Secondly, the primary goal of our work is to compute test inputs for possible anomalous cache behaviour in a single program.

A recent work [54] uses constraint-based test generation [5, 55] to partition the input domain of a program with respect to cache performance. Once all the partitions are computed, some manual interventions are required to locate the set of program locations that may exhibit issues related to cache performance. Besides, the work proposed in [54] computes a cache performance range for each partition. The cache performance range in [54] is computed via *static invariant generation* methods. As a result, the computed cache-performance range might be over-approximated, leading to *false positives*. Our approach, on the contrary, directly relates a cache thrashing scenario with the set of inputs (without any manual intervention). Moreover, since we generate test inputs based on dynamic analysis, our generated test-suite does not contain any *false positives*.

3.8 CHAPTER SUMMARY

In this chapter, we described a test generation framework that stresses the cache performance of a program. The key novelty in this technique is a systematic combination of static cache analysis and dynamic test generation via a set of instrumented assertions. Violation of any such assertion exposes a unique cache performance issue, specifically, a cache thrashing scenario in the program. As an output, our framework reports a test-suite where each test case in the test-suite points to a unique cache thrashing scenario along with a set of program inputs that leads to the same. Due to the use of dynamic test generation, our generated test-suite does not contain any spurious test cases. We have shown the application of our test generation framework in design space exploration and in cache performance optimization via cache locking.

4 ENERGY-CONSUMPTION ANALYSIS: BACKGROUND & LITERATURE REVIEW

This chapter introduces the reader to some the of key research directions on the topic of energy consumption analysis. In certain application scenarios, embedded systems are required to work in a mobile environment. Often such devices are powered by an on-board power-source with a finite capacity, such as a battery pack. To prolong the functional period of such energy-constrained devices, the programs running on such devices must be energy-efficient. This requirement of energy-efficiency creates a new set of challenges in the development and testing of programs intended for such systems. Existing research work have proposed a number of approaches to address these challenges. The research works which we shall discuss in this chapter are divided into four categories *i.e.* (i) techniques for average-case energy estimation, (ii) techniques for worst-case energy estimation, (iii) techniques for energy-aware testing and (iv) techniques for energy-aware programming.

4.1 ENERGY CONSTRAINED EMBEDDED SYSTEMS

Embedded systems are often used in applications that have real-time constraints. However, in certain application scenarios often energy constraints are the primary concern. For examples, a battery-powered sensor node deployed in a remote, inaccessible environment, that is required to collect and report data for weather monitoring purposes or the more ubiquitously used mobile devices such as smartphones, the usage for which is dictated by operational time between re-charges. Even though energy-efficiency is desired in all application scenarios, but it is specially crucial in the case of embedded devices that have limited amount of on-board battery power. There can be a number of approaches to ensure energy-efficiency in such systems. Some of the existing research works have presented techniques that assist in energy-aware design and development, while others propose the use of energy-aware testing and verification techniques. The choice of energy-consumption targeted technique however may depend on the application scenario itself. For instance, in the case sensor nodes, programs targeted at such systems must be analysed for the worst-case as well as average-case energy consumption, so as to make sure that the on-board battery is sufficient for the operational needs. Whereas, in the case of smartphones, using energy-aware development and testing techniques may suffice. In the following we shall discuss some of the existing research works on this topic.

4.2 APPROACHES USED FOR ENERGY TESTING/ESTIMATION

In the following sections, we shall discuss some of existing techniques that have been proposed on the topic of energy-consumption analysis. Most of the existing methods for energy analysis can be divided into the following four categories:

- *Estimating average-case energy consumption*: Some of the earliest works on energy-aware testing were proposed in this category. Such methods can be used to obtain

an estimate or an average-case, energy-consumption for a given program, on a given hardware, for a given input. Such methods can be further divided into two categories:

  - Architecture-based Energy Analysis: Techniques discussed in this category model the energy consumption of underlying hardware (such as processor, pipelines, etc) at varying levels of abstraction. An energy cost is associated with each operation that is conducted on a given hardware unit. Net energy consumption of a program is estimated as (approximately) the sum of energy-cost of all constituents instructions. Cycle-accurate simulators also fall under this category. These works are described in section 4.3.1.
  - Profiling-based Energy Analysis (section 4.3.2): These works, in general take the system (consisting of the hardware and software) as a black box and execute the given program for given set of inputs. The behaviour obtained as a result of execution (*i.e.* profile) is analysed for extracting appropriate information.

- *Estimating worst-case energy consumption*: As is the case with estimating worst-case execution time, worst-case energy-consumption analysis requires, static analysis based program flow analysis and micro-architectural modelling techniques. These works are described in section 4.4
- *Technique for detecting energy-inefficiencies*: These set of technique are mostly targeted at detecting suboptimal energy consumption behaviour in program. Techniques in this category may use static or/and dynamic analysis techniques. Energy-aware test generation techniques can also be found in category. These techniques are described in section 4.5.
- *Energy-aware programming*: Techniques in this category are primarily targeted at assisting the developer in developing energy-efficient programs rather that testing or validating them for energy efficiency. Such technique propose the use of energy-aware programming languages and energy-efficient programming constructs. These works are described in section 4.6.

4.3 ESTIMATING AVERAGE-CASE ENERGY-CONSUMPTION

Technique in this category can provide the approximate energy-consumption of a program, on a given hardware, (for a given test-input). In general. such techniques by themselves cannot be used to detect scenarios of sub-optimal energy-consumption behaviour or to estimate the worst-case energy consumption of a program (for a given hardware). In this section, such techniques are discussed in two parts: (i) Architecture-based energy analysis and (ii) Profiling-based energy analysis.

4.3.1 Architecture-based Energy Analysis

Techniques discussed in this category model the underlying hardware (such as processor, pipelines, etc) at varying levels of abstraction. An energy cost is associated with each operation that is conducted on a given hardware unit. Net energy consumption of a program is estimated as (approximately) the sum of energy-cost of all constituents instructions. These techniques can be further sub-divided into following three categories:

- Instruction Level Energy Analysis

- Cycle-accurate Simulators Based Energy Estimation
- Functional-block Level Energy Analysis

INSTRUCTION LEVEL ENERGY ANALYSIS Traditionally, estimation of energy consumption for processors, was performed at a very low level, which depended on detailed physical specification of the processor involved. Since this process is very cumbersome, the authors of [62] propose a instruction level modelling framework which can be applied to any off-the-shelf processor. The motivation behind their approach is that, given a instruction level power model for a processor and the program binaries (or assembly code), one should be able to determine the power consumption of the program on that processor. In order to obtain an instruction specific power model, they associated each instruction with a base energy consumption cost. To estimate the base cost of a particular instruction (say I), a program consisting of only instruction I is executed on the processor and resultant average energy consumption for I is measured. Although, this is a very simple way to estimate the base cost of an instruction but this approach might not be accurate. Inaccuracy might arise due to the fact that most modern processors have complex performance enhancing features (such as pipelines). Also note that the base cost for the same instruction may vary depending upon its operands. The authors argue that since the variation in the measured values is very small, average case values (for the base cost) can be used for most practical purposes. To compute the base energy consumption for the entire program, one needs to add-up the base energy costs of all the instructions in the program. The base energy consumption of a program is not equal to the actual energy consumption of the program, because a number of inter-instructions effects also influence the net energy consumption. Example of inter-instructions effects are the energy consumption due to the circuit state and energy consumed due to the hit or miss in the cache. The authors also found through their experiments, that the cost of executing a pair of instructions is always greater than the sum of the base cost of the instructions-pair. Authors term this cost as the circuit state overhead. Note that circuit state overhead can be potential source inaccuracy with their method. Another limitation and potential source of inaccuracy in their methods is the in-ability to model resource contention while executing instructions and other micro-architectural behaviour such as cache misses and branch mis-prediction.

Another such technique suggested by [63], uses a method of instruction level energy profiling for high performance RISC, embedded processors. Their techniques also assigns fixed, average-case energy value to each instruction, for all the instructions in the ISA of the architecture. But unlike the methods suggested by [62], they suggest that the average-case energy consumption for an instruction can be estimated by measuring the time it takes to execute that instruction (since energy = power $\times$ time). Through their experiments they observed that on an average, energy estimation by their method could lead to error of up to 8%. Also note that their methods does not take into account the variation in power consumption due to the inter-instruction effects which were mentioned in the above paragraph.

CYCLE-ACCURATE SIMULATORS This subcategory of energy analysis methods calculate the total energy consumption while executing a program, by performing a cycle-accurate simulation. SimplePower ([64]) and Wattch ([65]) are two such tools, which are based on methods for cycle-accurate power simulation.

SimplePower takes in a program (executables) as input and simulates it to generate a cycle-by-cycle energy estimate for that program. It also provides statistics for switch capacitance of the processor datapath, memory and on-chip buses using analytical energy models. The instruction set used in SimplePower is a subset of Simplescalar architecture ([66]) . The underlying architecture for SimplePower consist of a five stage pipeline, consisting of fetch, decode, execution, memory access and write back stages. The simulation method is very

straightforward, for each clock cycle, the tool simulates the execution of all active instruction and estimates the power for each active functional unit for that cycle. This simulation continues until the halt instruction is fetched, after which all the instructions in the pipeline are executed and the simulation stops. The tool uses a cache power simulator to simulates both the instruction as well as data caches. The tool also has bus simulator, which records the number of transitions on the bus. The bus statistics is combined with an interconnect power model to obtain the switch capacitance of the on-chip buses. A table for switch capacitance is maintained, which maps each input transition to a switch-capacitance. Note that the switch-capacitance table is architecture dependent. The table creation for this approach, is a tedious process because the size of table can grow exponentially with the number of input transitions. For complex modules such as memory, the tool uses an analytical, transition-independent model for energy estimation.

Wattch is another Simplescalar based tool for cycle-accurate performance estimation. It also provides cycle-accurate information for datapath elements, memory, control logic, and the clock distribution network. It is less expensive than the SimplePower, because it does not looks-up the switch-capacitance for each cycle. Instead, it keeps track of the number of access to a particular functional unit and scales it by average base-power dissipation for that functional unit, in order to calculate the total power consumption. Base power consumption for each functional unit is calculated before the analysis begins. The power estimates computed by Wattch may be less accurate than SimplePower, but Wattch has a very less computation overhead therefore it is more scalable for obtaining average-case energy consumption estimates of a program. However, it should be noted that neither Wattch not SimpleScalar, provide any mechanisms to estimate an upper bound on the energy consumption for a program execution.

FUNCTIONAL-BLOCK LEVEL ENERGY ANALYSIS A number of previous research works have demonstrated that the energy analysis of a program, can be performed at a much higher level of abstraction. Unlike the energy analysis methods mentioned in the previous categories, the methods of this category rely on functional decomposition of a systems, for power estimation. The key advantage of functional decomposition analysis is that, it very loosely coupled to processor architecture. For example, in order to estimate the power consumption of a program on a DSP, these methods would perform a functional analysis of the target DSP. Functional analysis of a DSP will include analysis with components such as control unit, memory management unit, etc.

One the first approaches for functional level energy analysis for embedded systems was proposed in [67]. In their approach, the power consumption of a functional unit is computed on the basis of a library of consumption rules. To build the library, a functional analysis of the targeted architecture is performed. All the components of the architecture which consume negligible power (for example, control registers for the DMA), are discarded. Some of the major high level components which are considered for functional decomposition are the control unit, memory management unit, instruction management unit and the processing unit. (Note that in the experiments the above mentioned functional units were sub-divided into smaller units) In order to induce the power consumption in a functional block, specially crafted code are executed on the DSP and the results are plotted on a set of charts. These charts are then used to identify parameters which influence the power consumption in a functional unit and finally, the library of consumption rules are created on the basis of these charts and parameters. The methods proposed in [68], [69], [70] have a similar approach.

4.3.2 Profiling-based Techniques

Techniques under this approach essential revolved around executing the program for set of input that generates the desired behaviour. While execution, power consumption can be estimated in a number of ways, each with different level of abstraction. One of the most straightforward ways would be measure the power consumption accrues the entire system. However, such a method may not provide enough information to deduce which components of the system cause the power consumption. Another approach is to measure the frequency of access to a given component during the execution and use these (frequency) number to estimate power consumption. (this is assuming that the power model for underlying hardware components are already available). [71] present one such work. In this work, the framework inserts *probes* or counters at appropriate program locations. These counters essentially measure the invocation of certain datapaths. Data obtained from these counters are then plugged in a library of existing empirical power model to obtain the net power consumption.

In recent times, due to the prevalent use of smartphone devices, topics related to functional and non-functional testing of smartphone applications have attracted the attention of software engineering research community. Recent works on energy-aware profiling [3, 72] have shown poor energy behaviour of several smartphone applications. In particular, the work in [3] present a energy-profiling technique for mobile apps. It also presented a few cases studies where it discusses the possibility of energy-inefficiency in a number popular mobile app (such as Angry Bird and Facebook). Another category of work [73] extends the idea of instruction level energy modeling to mobile devices. The key idea in [73], as well as earlier works such as [62], is to obtain a per-instruction, energy model for a given hardware system. The energy model associates each instruction with an energy consumption cost. One obvious complexity with such techniques is that the energy model is hardware-specific, therefore it must be recomputed every time the hardware changes. Additional complexities may arise (in creating the energy model), due to program-specific behaviour such as cache misses, branch mis-predictions, etc. Another recent work [74] has proposed a technique to relate power measurements with source lines of applications. Essentially the technique in [74] tries to map the power measurement data (obtain from the power-profile) to lines in the app source-code. To do so, it monitors the paths that are being executed while the profile is being recorded. Subsequently, it employs regression based techniques to map the path to estimated energy consumption, while accounting for high-level events such as thread-switches.

Since it is not possible to execute (or profile) an program for all possible inputs/configurations, it is not piratical to use profiling techniques such as the ones described in this section to estimate worst-case energy consumption of a program. We discuss the techniques for worst-cases energy consumption in the following section.

4.4 ESTIMATING WORST-CASE ENERGY CONSUMPTION

Most of the previous work on energy analysis, which was mentioned in the above few paragraphs (such as [62],[65],[67]), focusses on estimating the average-case energy consumption for a program. In particular, instruction level analysis methods are not capable of modeling the micro-architectural behaviour of architecture and may suffer from potential under-estimations. Therefore, such methods are unsuitable candidates for worst-case energy consumption (WCEC) analysis. Architectural level frameworks do not offer a good solution either, when it comes to worst-case energy consumption analysis. Because in order to obtain WCEC, such methods would be required to simulate the program, on it's entire input space, which is clearly infeasible. For similar reasons, functional level frameworks, also do not offer

a good solution for obtaining the WCEC for real-time, embedded systems. In the following subsection, we will describe some of the static analysis based methods which have been used for estimating the WCEC of a program.

One of methods described in section 2.4 ([63]), uses average-case execution time of an instruction to estimate the average-case execution energy of that instruction. So with the same intuition, one might wonder, if the worst case execution time of a program can used to estimated the worst case energy consumption of that program. Interestingly, experiments by [75] revealed that the worst case energy path for a program does not necessarily coincide with the worst case execution time path. They suggested that such un-intuitive behaviour, is observed because the switching activity in the processor circuit, may not be directly related to the execution time of the program. Based on this observation they proposed one of the first techniques for WCEC estimation of a program. They split the WCEC analysis into two parts: time dependent and time independent analysis. The authors classify the instruction specific energy analysis as a time-independent component whereas the time-dependent analysis consists of component such as pipeline specific energy analysis. The reason for such a classification, they explain, is that the energy spent on various hardware components such as switch-off power, clocking circuit, leakage power can not be attributed to any particular instruction and it is roughly proportional to the execution time. Therefore, the energy of basic block ($Energy_b$), can calculated as

$$Energy_b = Dynamic_b + SwitchOff_b + Leakage_b + Clock_b$$

where the $Dynamic_b$ represents the instruction specific energy consumed of the basic block and $SwitchOff_b$, $Leakage_b$, $Clock_b$ represents the switch-off power, leakage power and clock circuit power respectively, during the execution time of the basic block b. Leakage power is the term used to denote the unintended power loss in the processor. The rate of leakage power depends on the processor technology but it is usually a constant for a given architecture.

Most modern processor employ some mechanism to switch-off the unused portions of the circuit, when they are not in use. Ideally, a component should draw peak power when it is in use and no power when it is not used. But the observed behaviour is somewhat different, even when an unit is switched-off it dissipates some power. The power consumed by a component in switched-off state is termed as switch-off power. Some approaches assume that a (multi-ported/single-ported) component (such as a register file) would consume a peak power even if there is a single access to the unit (this approach is also known as simple clock gating mechanism). Other approaches, such as realistic clock gating mechanism assume that, even in the switched-off state components can consume up to 10% of their peak power consumption. The techniques presented by [75], uses a combination of these approaches. It assumes that the peak power is consumed by a component c, while executing a basic block b, for $min(access_b(c), wcet_b)$ cycles. Here $wcet_b$ refers to the worst case execution time of basic block b. For the switch-off power estimation, it assumes that 10% of the peak power is consumed by a component in the switch-off state. So the switch-off power for a component c, while execution of basic block b, can be written as

$$SwitchOff_b(c) = \{WCET_b - \frac{Access_b(c)}{Ports_c}\} \times PeakEnergyConsumption_c \times 10\%$$

The processor model used for this work is similar to the Simplescalar and has a five-stage pipeline (Fetch-Decode & Dispatch-Execute-Write Back-Commit). Instruction are executed in an out-of-order fashion, but they are fetched, decoded and committed in program order.

In the above equation for energy calculation of a basic block, the $Dynamic_b$ component represents the sum of energy spent by all the instructions in a basic block. The $Dynamic_b$ for a basic block is affected by a number of factors such as, energy consumption for register access, energy consumed in the circuit-selection logic and the energy consumed in the wake-up [1] logic. The authors assume that the energy consumed in the register files is proportional to the number of registers used in the basic block. The wakeup logic energy consumption is assumed to be proportional to the number of output variables produced in the basic block. The selection logic is assumed to be accessed in every cycle. To estimate the WCEC of the entire program, the constraints are represented as ILP problem, as done by [35]. The ILP formulation is also used to capture the effects of cache and branch prediction behaviour, along with the program flow analysis. The objective function represents the total energy consumed by the program. So the WCEC can be estimating by maximizing the objective function.

4.5 DETECTING ENERGY-INEFFICIENCY

Techniques based on static analysis: Existing works have proposed program analysis based techniques to uncover energy inefficiencies in mobile apps. Recent works such as [76, 77] propose static analysis based techniques for detecting resource leaks in Android apps. Static analysis based resource leak detection techniques have been proposed for Java programs as well [78]. Such works, in general, try to verify that resources that are acquired during the execution of the program are released along all paths leading to exit(s). The idea being that the certain resources are energy-intensive, therefore, should be released by the program, before it ceases to execute. [76] in particular, formulates the resource leak detection problem as a reaching definition problem. It essentially analyses all program-paths where an energy-intensive resource is acquired or released in the program, using data-flow analysis. However, the technique describe in [76] may be limited in finding resource leaks due to the representation (of mobile apps) it uses. One of the key challenges in analysing mobile apps arises due to the fact that they are event-driven applications. What this implies is that such programs do not have an explicit *main* method, instead, what is present is a set of event-handler, each one programmed to process pre-defined set of events. Since the ordering of arrival of events is unknown (events come from the environment) apriori, actual execution of event handlers is also unknown. The work of [76] does not address this important challenge, instead it constructs aggregate CFG, consisting of several smaller CFGs (from event handlers). How the ordering of these event-handlers is obtained, is not explained. However, one possible source for obtaining such information could be profiling. [76] also mentions the possible use of developer-assistance to bypass this challenge. The work of [77] uses a similar approach based on static analysis, for resource leak detection. However, it does tries to address the challenge of analysing event-driven applications, such as mobile apps. Essentially, it uses a data-structure which associates events to respective event-handlers, for a given activity. As a result, knowing the ordering between event-handlers beforehand, may not be necessary. One common drawback with static analysis based approaches, such as the ones mentioned in this paragraph, is that they may generate false positive (as observed in [77]). This may be due to the presence of infeasible program-paths within the app source code.

Techniques based on dynamic analysis: A number of works , such as [79, 80], have proposed techniques based on dynamic program analysis to estimate the energy consumption of a program. For instance, the work of [79] uses symbolic execution along with platform specific energy profiles to generate estimated energy-consumption along a given (explored)

1 The energy required to re-start dependent instructions, when all of their dependencies are fulfilled

program-path. The energy-profiles essentially contain the energy-consumption profiles of each basic-block in the program. To compute the energy consumption for a given path, the energy consumption for a given block is multiplied by the counter for each basic block, on a given path. The counter essentially counts the number of times a given basic block is executed for a given execution. Such an approach can be considered preliminary in the sense that it only considers the CPU power consumption. In contrast, power consumption to access memory subsystems, network card and other I/O components were not considered. In smartphone devices, I/O components consume the most power, hence this technique may not be very suitable for analysing mobile apps. The work of [80] is more suitable for testing the energy-consumption behaviour of mobile-apps. It dynamically explore the graphical user interface model of the subject app to detect the presence of resource leaks. A common limitation to the dynamic analysis method as described in preceding paragraphs is that these method need to explore all program paths in the program in order to produce complete results. This however, may be impractical for many program which may have explicit loops(for, $while$, $do-while$) or implicit loops (due to cycles in GUI model). This issue (commonly referred to as state-space-explosion problem) may also lead to scalability issues while testing real-life programs.

Test generation for mobile apps: Works related to test generation in mobile apps have mostly been confined to the domain of functional testing. Works such as [81, 82] have proposed techniques to test the functional properties of mobile apps. [81] in particular uses a biased-random testing technique to explore the various GUI states of an app, whereas [82] uses symbolic execution to achieve the same goal.

As of this writing, there were two different approaches, [2] and [83], targeted at detecting energy-inefficiencies in mobile apps. One of the approaches is our previous work [2] which uses a hardware-software hybrid approach to systematically generate test inputs that leads to energy-inefficient scenarios. On a high-level, the technique of [2] can be described as a grey-box testing approach where real-time measurements from the device are used to detect energy-inefficiencies in app, that is being executed on the device. Due to the use of real-power measurements, the technique of [2] of can skip the expensive, model-generation stage. Also using real measurements instead of power model further reduces the number of false-positives in final results.

Another approach for generating energy-consumption related test cases was proposed in a recent work of [83]. The technique in [83], unlike the technique of [2], can be described as a white-box testing based approach, which uses bounded symbolic execution to detect resource leaks in mobile apps. Bounded symbolic execution is an attempt to bypass the problem of state-space-explosion. As explained in the previous paragraph, any technique which relies on exploring all program path within a program may have scalability issues. This is because in the presence of unbounded exploration there may be infinite number of feasible program paths to explore. Using bounded symbolic execution alleviates this issue but may introduce new limitations. For instance, bounded symbolic execution may be unable to detect feasible resource leaks, if the bounds (for the bounded exploration) are set too cautiously.

To provide a complete solution to the requirement of detecting, validating (generating test cases) and repairing resource leaks in mobile apps, we have developed a framework which is described in Chapter 6. This technique differs from the existing two directions of works (as described in previous paragraphs) in three aspects (i) the way energy inefficiencies are detected (power measurement vs static analysis) (ii) the way test-cases are generated (search heuristics vs guided symbolic execution) and (iii.) automatic repair expressions generation (other test-generation technique such as [2] and [83] may require manual effort).

4.6 ENERGY AWARE PROGRAMMING

A different line of work aims to produce energy-efficient applications from different implementations of the same functionality [84, 85]. Specifically, in application scenarios where fast but approximate answer is acceptable, such approaches could be quite useful. For example, while compressing files, there is always a trade-off between the achieved compression rate and time required for compression. Likewise, many applications scenarios, can benefit by using approximate, energy efficient computing. The decision to choose an implementation is influenced by monitoring the power consumption for a *given test-suite*. For instance, the work in [84] dynamically chooses approximate implementations of an energy-intensive functionality (such as long-running loops), to reduce the power consumption. The framework can dynamically approximate the resource expensive loops and functions depending upon the requirements, while maintaining a pre-defined, minimum quality of service(QoS). Through their framework, a programmer can provide a minimum required QoS, along with multiple approximate versions of the same function, for function approximation. Likewise loops approximation can be achieved by running the loop for a fewer number of iterations. The framework can calculate the loss in QoS, to determine the best approximation to be used for a particular scenario. Along the same lines, a recent work [85] monitors the power consumption of different API implementations and computes the potentially best implementation in terms of energy-efficiency. Specifically, given a program that uses Java Collection Framework, the technique of [85] automatically generates several alternative versions of the same program by replacing a Java-collection object by another object of similar behaviour. Subsequently, it executes this alternate versions to find the most energy-efficient alternative.

A complimentary approach to energy-aware programming has been proposed by [86]. The work of [86] proposes a new language consisting of novel type system, Energy Types. In general, non-functional properties, such as energy-consumption, are not specifically encoded in the program source-code. As result of which testing and verification of energy-constraints is challenging. With this kind of type system, not only the task of energy-aware testing and verification is much easier, but it also enables the programmer to encoded their assumption and expectation about energy-consumption of the program within itself. For instance, exploiting the information provided by the programmer, energy saving mechanisms such as dynamic voltage and frequency scaling can be used aggressively.

4.7 CHAPTER SUMMARY

In this chapter we discussed four important direction of work on the topic of energy-consumption analysis, which are, (i) techniques for average-case energy consumption, (ii) techniques for worst-case energy consumption, (iii) techniques for detecting energy-efficiencies and (iv) techniques for energy-aware programming. In general, average-case energy consumption techniques can be light-weight and fast (such as profiling techniques) or accurate and slow (such as cycle-accurate simulators). However, these techniques cannot be employed for estimating worst-case estimation. Techniques for detecting energy-inefficiencies in program comes in may flavours. There exists static analysis based techniques which verify the absence of energy-intensive resource leaks in programs and there exists symbolic execution based techniques that can give per-path energy consumption cost for a given program. Finally, we looked at some of the existing works on energy-aware programming and optimization. Such works either provide novel programming constructs for energy-aware programming or provide technique to automatically optimize the energy-consumption of a given program for the given input space.

5 DETECTING ENERGY BUGS AND HOTSPOTS IN MOBILE APPS

Over the recent years, the popularity of smartphones has increased dramatically. This has lead to a widespread availability of smartphone applications. Since smartphones operate on a limited amount of battery power, it is important to develop tools and techniques that aid in energy-efficient application development. Energy inefficiencies in smartphone applications can broadly be categorized into *energy hotspots* and *energy bugs*. An *energy hotspot* can be described as a scenario where executing an application causes the smartphone to consume abnormally high amount of battery power, even though the utilization of its hardware resources is low. In contrast, an *energy bug* can be described as a scenario where a malfunctioning application prevents the smartphone from becoming idle, even after it has completed execution and there is no user activity. In this chapter, we present an *automated test generation framework* that detects energy hotspots/bugs in Android applications. Our framework systematically generates test inputs that are likely to capture energy hotspots/bugs. Each test input captures a sequence of user interactions (*e.g.* touches or taps on the smartphone screen) that leads to an energy hotspot/bug in the application. Evaluation with 30 freely-available Android applications from Google Play/F-Droid shows the efficacy of our framework in finding hotspots/bugs. Manual validation of the experimental results shows that our framework reports reasonably low number of false positives. Finally, we show the usage of the generated results by improving the energy-efficiency of some Android applications.

5.1 NEED FOR AUTOMATED ENERGY-AWARE TEST GENERATION

Global penetration of smartphones has increased from 5% to 22% over the last five years. As of 2014, more than 1.4 billion smartphones are being used worldwide [87]. Over the recent years, smartphones have improved exponentially in terms of processing speed and memory capacity. This improvement has allowed application developers to create increasingly complex applications for such devices. Additionally, modern smartphones are equipped with a wide range of sensors and I/O components, such as GPS, WiFi, camera, and so on. These I/O components allow developers to create a diverse set of applications. In spite of such high computation power and developer flexibility, the usage of smartphones has been severely impeded by their limited battery capacity. In terms of computation capacity, most of the current-generation smartphones are two or even three orders of magnitudes better than their decade-old counterparts. However, the battery-life of these modern smartphones has improved only two or three times[1]. *High computational power coupled with small battery capacity and the application development in an energy-oblivious fashion can only lead to one situation: short battery life and an unsatisfied user base.*

Energy inefficiencies in smartphone applications can broadly be categorized into *energy hotspots* and *energy bugs*. An *energy hotspot* can be described as a scenario where executing an application causes the smartphone to consume abnormally high amount of battery power even though the utilization of its hardware resources is low. In contrast, an *energy bug* can be

1 For instance, if we compare `Nokia 9000 Communicator` (released in 1996) to `Samsung S3` (released in 2012), we can observe that the processing power has increased from `24MHz` to `1.4GHz`, whereas the battery capacity has only increased from `800mAH` to `2100mAH`

described as a scenario where a malfunctioning application prevents the smartphone from becoming *idle even after it has completed execution and there is no user activity*. Table 5 lists the different types of *energy bugs* and *energy hotspots* that can be found in Android applications. It is also worthwhile to know that most contemporary smartphone devices are designed to operate at different power states and prolong the battery life. However, as listed in Table 5, malfunctioning applications may lead to inappropriate power states, such as energy hungry GPS/sensor updates, *non-idle* power state in the absence of user activity and so on. Moreover, most of these energy inefficiencies appear when the application does not access the device resources in an appropriate fashion (*e.g.* not releasing WiFi/GPS/Wakelocks or expensive sensor updates), eventually hampering the battery life. Therefore, to build energy-efficient applications, it is crucial for the developer to know these energy inefficiencies in the application code. Presence of such energy inefficiencies in the application code can be highlighted to the developer via our proposed methodology.

In this chapter, we present an automated test generation framework to detect energy hotspots/bugs in Android applications. Specifically, our framework systematically generates test inputs which are likely to capture energy hotspots/bugs. Each test case in our generated test suite captures a user interaction scenario that leads to an energy hotspot/bug in the respective application. We argue that the systematic generation of such user interaction scenarios is challenging. This is primarily due to the absence of any non-functional property (*e.g.* energy consumption) annotations in the application code. As a result, any naive test-generation strategy may either be *infeasible* in practice (*e.g.* exhaustive testing) or it may lead to an extremely poor coverage of the potential energy hotspots/bugs. This also brings us to the difficulty of defining an appropriate coverage metric for any test generation framework that aims to uncover energy hotspots/bugs. In our framework, we address these challenges by developing a directed search strategy for test generation.

To design a directed search strategy, it is critically important to know the potential sources of undesirable energy consumption. Table 5 lists such sources of energy consumption in Android applications. Moreover, existing works such as [3] have shown that I/O components are primary sources of energy consumption in a smartphone. One crucial observation is that I/O components are usually accessed in application code via *API calls*.

Besides, the power management functionality (*e.g.* Wakelocks), background services and other hardware resources (*cf.* Table 5) of a device can only be accessed through a set of API calls. In summary, most of the classified energy hotspots/bugs (*cf.* Table 5) are exposed via the invocation of API call(s). Therefore, the general intuition behind our directed search strategy is to systematically generate user interaction scenarios which potentially invoke such API calls.

Our search strategy revolves around systematically traversing an event flow graph (EFG) [88]. EFG is an abstraction to capture a set of possible user interaction sequences. Each node in an EFG captures a specific user interaction (*e.g.* touching a button on smartphone screen), whereas an edge in the EFG captures a possible transition between two user interactions. Therefore, each trace in an EFG captures a possible sequence of user interactions. Since exhaustive enumeration of EFG traces is potentially infeasible, our directed search methodology generates appropriate EFG traces which are likely to lead to undesirable energy consumption. To accomplish this, we primarily employ two strategies. Based on our observation from Table 5, we execute selected EFG traces and these selected EFG traces invoke API calls that might be responsible for irregular power consumption . Besides, if an energy hotspot/bug is detected after executing an EFG trace, we record the sequence of API calls responsible for such irregular energy behaviour. Subsequently, we prioritize unexplored EFG traces that may invoke a similar sequence of API calls. Such a guidance heuristic primarily aims to uncover as many energy hotspots/bugs as possible in a limited time budget.

Table 5: Classification of Energy Bugs and Energy Hotspots

#	Category	Energy Bug	Energy Hotspot
a	Hardware Resources	*Resource Leak*: Resources (such as the WiFi) that are acquired by an application during execution must be released before exiting or else they continue to be in a high-power state [89]	*Suboptimal Resource Binding*: Binding resources too early or releasing them too late causes them to be in high-power state longer than required [90], [91]
b	Sleep-state transition heuristics	*Wakelock Bug*: Wakelock is a power management mechanism in Android through which applications can indicate that the device needs to stay awake. However, improper usage of Wakelocks can cause the device to be stuck in a high-power state even after the application has finished execution. This situation is referred to as a Wakelock bug [92]	*Tail-Energy Hotspot*: Network components tend to linger in a high power state for a short-period of time after the workload imposed on them has completed. The energy consumed by the component between the period of time when the workload is finished and the component switches to the sleep-state is referred to as Tail Energy [93]. Note that tail energy does not contribute to any useful work by the component. Scattered usage of network components throughout the application code increases power loss due to Tail-Energy
c	Background Services	*Vacuous Background Services*: In the scenario where an application initiates a service such as location updates or sensor updates but doesn't removes the service explicitly before exiting, the service keep on reporting data even though no application needs it [94]	*Expensive Background Services*: Background services such as sensor updates can be configured to operate at different sampling rates. Unnecessarily high sampling rate may cause energy hotspots and therefore should be avoided. [95] Similarly, fine-grained location updates based on GPS are usually very power intensive and can be replaced by inexpensive, WiFi-based coarse-grained location updates, if an application is using both the WiFi and the GPS [96]
d	Defective Functionality	*Immortality Bug*: Buggy applications may re-spawn when they have been closed by the user, thereby continuing to consume energy [97]	*Loop-Energy Hotspot*: Portions of application code are repeatedly executed in a loop. For instance, a loop containing network login code may be executed repeatedly due to reasons such as unreachable server [97]

Besides the challenges encountered in generating energy stressing test inputs, it is also *non-trivial* to *automatically* detect a potential energy hotspot/bug in a given trace. To detect energy hotspots/ bugs, our framework executes a test input (*i.e.* a user interaction scenario) on a off-the-shelf smartphone, while simultaneously measuring the power consumption via a power meter. To detect an energy bug in a specific trace, we measure the statistical dissimilarities in power-consumption trace of the device, specifically, before and after executing the respective application. As the power consumption of an idle device should be similar, a statistical dissimilarity indicates an *energy bug*. To detect an energy hotspot, we employ an anomaly detection technique [98] to locate anomalous power consumption patterns. Once we finish the process of detecting hotspots/ bugs in a power-consumption trace, we generate a different user interaction scenario (using the directed search strategy in the EFG) to investigate. The test generation process continues till the time budget permits or all event traces invoking API calls have been explored. As the API calls are the potential locations to cause irregular energy behaviour, the quality of our test suite is provided via the coverage of API calls in the application.

5.2 GENERAL BACKGROUND

Android is an open-source operating system (OS) designed for mobile devices such as smartphones. We choose Android as our target platform primarily due to its relevance in the real world (globally 57% of all smartphones/tablets are Android based [99]). Additionally, a wide variety of tools are publicly available for Android application developers. This includes, among others, tools to monitor the state of an application in real-time (*e.g. logcat*), to communicate with the device (*e.g. android debug bridge*) and to facilitate application development and testing (*e.g. emulator*).

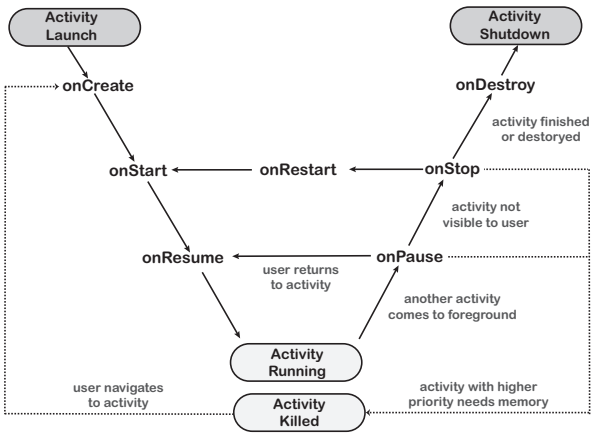


Figure 21: Life-cycle of an Android activity

The user interaction interface of an Android application is referred to as an *Activity*. Figure 21 shows the life-cycle of an Android activity. An activity can be in one of the seven stages during its life-cycle. Usually, all the set-up tasks (such as acquiring resources and starting background services) take place in four stages of the activity, namely *onCreate*, *onStart*, *onResume* and *onRestart*. Similarly, all the tear-down tasks (such as releasing resources and stopping background services) take place in three stages, namely *onPause*, *onStop* and

onDestroy. However, some real-life applications do not follow the ideal set-up and tear-down scenarios as explained via Figure 21. Such applications may contain *energy bugs*. This situation is made worse by the fact that most real-life applications have a huge number of feasible user interaction scenarios (due to complex GUIs). As a result, it can be impossible for a developer to test an application for all possible scenarios.

```
1  LocationManager locationManager;
2  long Min_Update_Time = 10, Min_Distance = 1000 * 60 * 1;
3
4  @Override
5  public void onCreate(Bundle savedInstanceState){
6      super.onCreate(savedInstanceState);
7      setContentView(R.layout.main);
8      locationManager = (LocationManager)getSystemService
9                                        (LOCATION_SERVICE);
10     locationManager.requestLocationUpdates
12     (LocationManager.GPS_PROVIDER,Min_Update_Time,
13                               Min_Distance, this);
14     someOtherFunctionality();
15 }
16
17 @Override
18 public void onPause(){
19     super.onPause();
20     try{
21           functionMayThrowsException();    <---------
22           locationManager.removeUpdates(this);
23     }catch(Exception ex){
24           Log.v("test","exception occured");
25     }
26 }
```

Figure 22: Code with a potential energy bug

Figure 22 shows a snippet of application code that has a potential energy bug. The application code is supposed to start a location-update background service (Line 10) in the *onCreate* method. Subsequently, it performs some operation with list data (Line 12). When the user stops the application, the location-update service is removed (Line 19) in the *onStop* method. However, if there is an exception before Line 19 (for instance, due to Line 18), the location update service is never stopped, resulting in an *energy bug*. The example in Figure 22 shows one possible scenario (*cf.* Table 5 #c: *Vacuous Background Services*) which can lead to an energy bug. Next, we shall show an example that can lead to an *energy hotspot*.

The code snippet in Figure 23(a) shows an example with energy hotspots due to disaggregated network activities (*cf.* Table 5 #b: *Tail-Energy Hotspot*). Observe that in Figure 23(a), network related code (Line 6) is interleaved with CPU-intensive code (Line 8) within the same loop. Such an interleaving causes energy-inefficiencies due to *Tail-Energy* (see Table 5 #b: *Tail-Energy Hotspot*). *Tail-Energy* behaviour has been observed for network components such as *3G*, *GSM* and *WiFi* [93]. Other works [3] have observed *Tail-Energy* in components such as storage disks and *GPS* as well. In order to reduce energy-loss due to *Tail-Energy*, the network related code in Figure 23(a) can be aggregated as shown in Figure 23(b).

Finally, we shall explain the method used for obtaining the power consumption ratings of the hardware components in our smartphone. One approach to obtain the power consumption ratings would be to perform empirical experiments based on the guidelines provided on

```
1  public Object[] nonAggregatedComm()
2  {
3    Object[] objectArray =
4                   new Object[10];
5    for(int i=0; i<10; i++){
6      Object temp = downloadObject(i);
7      objectArray[i] =
8          processObject(temp);
9    }
10   return objectArray;
11 }
12
13
```

(a)

```
public Object[] aggregatedComm()
{
  Object[] tempArray = new Object[10];
  for(int i=0; i<10; i++){
    tempArray[i] = downloadObject(i);
  }
  Object[] objectArray = new Object[10];
  for(int i=0; i<10; i++){
    objectArray[i] =
        processObject(tempArray[i]);
  }
  return objectArray;
}
```

(b)

Figure 23: (a) Code with energy hotspot due to disaggregated communication (b) Code without energy hotspot

the Android developer web page [100]. However, there is a more elegant way to obtain the power consumption ratings. Most Android smartphones are shipped with a XML file (usually named as *power_profile.xml*) containing the average power consumption ratings for the hardware components in the device. The data contained in this XML file is provided by the device manufacturer and therefore it is reliable. Moreover, the Android framework uses this data to show battery related statistics. However, note that the data in this XML file is an indicator of average power consumption of the hardware components of the device and does not correspond to any particular application being run on the device. The data from *power_profile.xml* for our smartphone `LG L3 E400`, is shown in Figure 24.

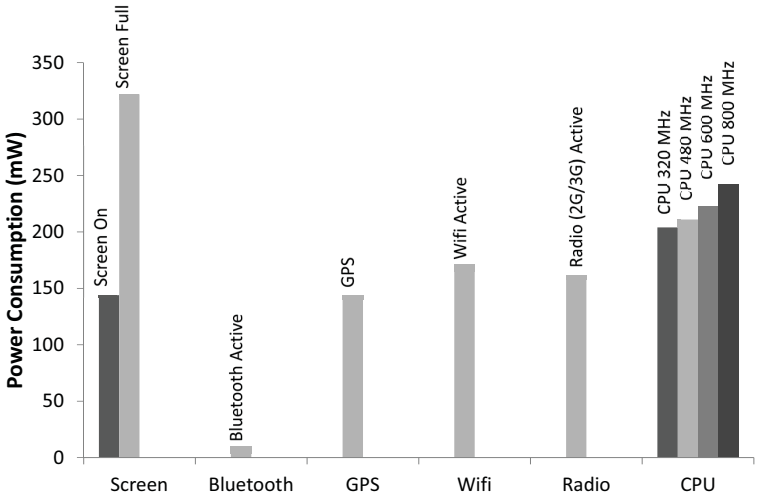


Figure 24: Power profile for LG Optimus L3 E400 smartphone

5.3 DETECTING ENERGY BUGS AND HOTSPOTS IN MOBILE APPS: AN OVERVIEW

An overview of our test-generation framework is shown in Figure 25. Our framework has two essential components: *(i)* guided exploration of selected event traces that are more likely to uncover energy hotspots/bugs, and *(ii)* detection of hotspots/bugs in a given event trace for an application. The information provided by the hotspot/bug detection component is also utilized by the guidance component to select subsequent event-traces. The process of selection, execution and detection continues until the given time-budget has expired or all event-traces invoking API calls have been explored. Finally, event traces that lead to energy hotspots/bugs are reported to the developer for further investigation.

To detect a hotspot/bug, we measure the power consumption of the application for a given event-trace. However, it is impossible to detect a hotspot/bug in an application solely by analyzing its power consumption trace. For instance, consider a scenario where two programs P_1 and P_2 have similar power consumption traces. However, program P_1 has a much higher utilization of system resources (such as CPU) compared to P_2. In such a scenario, program P_1 is more energy-efficient than program P_2. Therefore, to accurately detect energy inefficiencies, it is important to define an appropriate metric for system-resource utilization.

For a hardware component x, the $Load_x$ represents the average amount of computational work performed by the hardware component x over a given period of time. $Load_x$ has a range from 0 to 1. For example, $Load_{CPU}$ represents the fraction of time CPU is in use and therefore $Load_{CPU}$ can be a number between 0 and 1. For other hardware components (*i.e.* WiFi, screen, Radio and GPS), $Load_x$ captures whether the respective components are in use. For instance, $Load_{WiFi}$ is set to 1 if the WiFi is transmitting data and it is set to 0 otherwise. For any hardware component x, we measure $Load_x$ directly from the device, while the application under test is being executed. It is important to note that a higher $Load_x$ in a high-power consuming component x would result in a higher power consumption for the device. Based on this information we define a new metric of *utilization* that will be subsequently used in energy hotspots/bugs detection.

Definition 5.3.1 *Utilization (U) can be defined as the weighted sum of utilization rates of all major power consuming hardware components in a device, over a given period of time.*

Based on the power profile for our device (*cf.* Figure 24), major power consuming components in our mobile device are the screen, WiFi, Radio, GPS and CPU. Therefore, for a given time interval, the utilization of system resources can be computed by Equation 32.

$$\text{Utilization} = U_{Screen} + U_{CPU} + U_{WiFi} + U_{Radio} + U_{GPS} \tag{32}$$

$$U_{CPU} = \begin{cases} W_{CPU_{320}} \cdot Load_{CPU}, & \text{if CPU is operating at 320MHz} \\ W_{CPU_{480}} \cdot Load_{CPU}, & \text{if CPU is operating at 480MHz} \\ W_{CPU_{600}} \cdot Load_{CPU}, & \text{if CPU is operating at 600MHz} \\ W_{CPU_{800}} \cdot Load_{CPU}, & \text{if CPU is operating at 800MHz} \end{cases}$$

$$U_{Screen} = \begin{cases} W_{Screen_{ON}} \cdot Load_{screen}, & \text{if screen on} \\ W_{Screen_{FULL}} \cdot Load_{screen}, & \text{if at full brightness} \end{cases}$$

$$U_x = W_x \cdot Load_x, \quad x \in \{WiFi, Radio, GPS\}$$

In Equation 32, U_x represents the utilization of hardware component x. Utilization of a component x is directly proportional to its $Load_x$. For any component x, the value of W_x is computed from the power profile (Figure 24). Specifically, the value of W_x is normalized

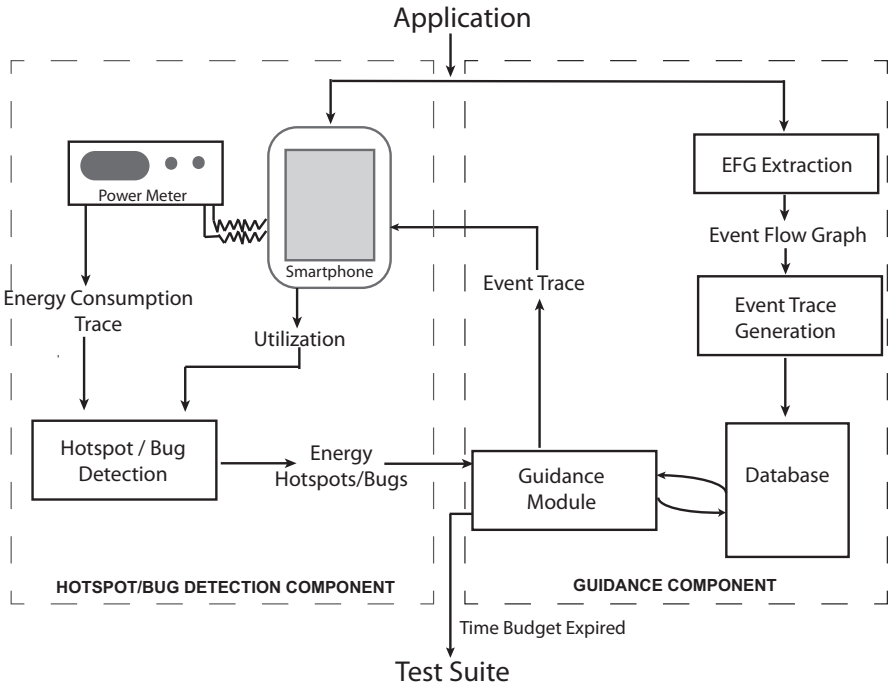


Figure 25: Overview of the test generation framework

such that W_x for the most power consuming component is 1 (in our case $Screen_{FULL}$ as shown in Figure 24). Note that in our case Equation 32 does not include Bluetooth. This is because in our target device Bluetooth has a very low power consumption compared to other components. However, if required, we can easily extend Equation 32 to accommodate Bluetooth as well. Using the new metric of utilization (U), we can now compute the magnitude of energy-inefficiency as follows.

Definition 5.3.2 *Energy-consumption to Utilization (E/U) ratio is the measure of energy-inefficiency of an application for a given time period.*

If E/U ratio of an application is high, it implies that the energy-consumption is high, while utilization is low. Therefore, a high E/U ratio indicates that the application is *energy-inefficient*. Recall that we discuss two categories of energy issues that can make an application energy-inefficient *i.e.* energy hotspots and energy bugs. They can be defined as follows: A high E/U ratio during the execution of an application indicates the presence of an energy hotspot. A persistently high E/U ratio even after the application has completed execution indicates the presence an energy bug.

Now we shall briefly discuss the exploration of event traces to reveal hotspots/bugs. In our framework, guided exploration of selected event traces is based on event flow graph (EFG) [88]. EFG of an application can be defined as follows.

Definition 5.3.3 *An Event Flow Graph (EFG) is a directed graph, capturing all possible user event sequences that might be executed via the graphical user interface (GUI). Nodes of an EFG represent GUI events. A directed edge between two EFG nodes X and Y represents that GUI event Y follows GUI event X.*

In our experiments, we use a modified version of the Dynodroid tool [81] to generate the EFG. Subsequently, our framework generates event sequences up to maximum length k and

stores them in a database. After the event traces have been generated, our framework initiates a guided exploration of those traces. The crucial factor during the exploration is to identify the event traces that may lead to hotspots or bugs. Our framework accomplishes this by selecting event traces based on the number of invoked API calls and guidance heuristic. The guidance heuristic gathers information from previously detected hotspots/bugs, specifically the sequence of API calls which are likely to lead to energy inefficiencies. Subsequently, the selection process is biased towards event traces invoking a similar sequence of such API calls. This process of selection, execution and detection continues until the time-budget has expired or all event-traces invoking API calls have been explored. Finally, event traces that lead to energy hotspots/bugs are reported to the developer for further investigation.

5.4 DETAILED METHODOLOGY

In the following sections, we shall describe our test generation methodology in detail. Broadly, our framework contains two substeps; *(i)* preprocessing the application under test to build a database of possible event traces, and *(ii)* test generation using event traces generated in the first step.

5.4.1 Preprocessing the Application

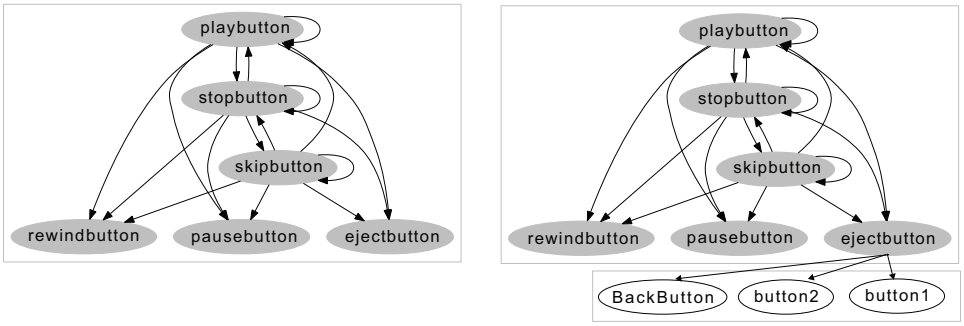
Preprocessing of application can be divided into three steps: (i) EFG extraction (ii) Event trace generation (iii) Extraction of API calls sequence for each event trace. Note that this preprocessing step is performed only once for each application. The generated EFG and database are stored for later use and need to be updated *only if* the application's user interface changes. Since this preprocessing is done *offline*, a developer can rerun the test generation step (detailed in Section 5.4.2) without repeating preprocessing step.

(i) Event Flow Graph Extraction : We build the Event Flow Graph (EFG) based on the UI model proposed in [88]. For the purpose of EFG construction we use two third-party tools Hierarchy Viewer[101] and Dynodroid[81]. Hierarchy Viewer provides information about the UI elements of the application under execution and Dynodroid is used to explore these event sequence automatically. Note that Dynodroid does not generate the EFG by itself, therefore we modified the Dynodroid source code to build the EFG. The EFG was constructed gradually each time Dynodroid interacts with the application. Figure 26 shows how our EFG is being gradually built as Dynodroid performs the exploration of event sequences. It is worthwhile to note that Dynodroid does not guarantee to reach *all* GUI states during exploration. Therefore, our constructed EFG is in fact a *partial* EFG of the entire application. However, in our experiments, we observed that the generated EFGs cover most of the GUI elements for the tested applications.

(ii) Event Trace Generation : EFG is primarily used to generate a set of event traces. Note that each application has a start GUI screen. This GUI screen is presented to the user when an application is launched. We refer to this GUI screen as the *root screen*. Therefore, for a sequence of user interactions performed in an application, the first action corresponds to an event present in the *root screen*. Using this notion, we define an *event trace* as follows.

Definition 5.4.1 *An event trace is defined as a path of arbitrary length in the EFG. Such a path must start from an event in the root screen of the respective application.*

Based on our EFG, we generate a complete set of event traces upto length k. These event traces are stored in a database for further analysis during test generation. Figure 26(b) shows the partial EFG of an application. The node containing the event *playbutton* captures the *root screen* of the same application. An example *event trace* of length 3 would be *playbutton* $\rightarrow$ *stopbutton* $\rightarrow$ *playbutton* or *skipbutton* $\rightarrow$ *ejectbutton* $\rightarrow$ *BackButton*. Note that events *playbutton* and *skipbutton* correspond to different events in the *root screen* of the application.



(a) (b)

Figure 26: (a) An example EFG (b) EFG after pressing "ejectbutton"

(iii) Extraction of API Calls : Existing literature [3] has shown that I/O components are one of the major sources of energy consumption in smartphones. On observing the power profile of our smartphone (see Figure 24) we find this argument to be consistent. In general, for modern smartphones the major power consuming components are the screen, CPU, WiFi, Radio, GPS, SDCard, Camera and Audio hardware. We observed that these components (except for the CPU) can only be accessed via a set of API calls provided by the Android SDK framework. Therefore, we create a pool of such API calls. Table 6 shows a categorization of these API calls based on their functionalities. Since our target device (`LG L3 E400`) uses Android 2.3 (Gingerbread), therefore we only consider API calls available in Android 2.3. It is worthwhile to note that such a pool is constructed *only once* and it needs to be updated *only if the Android SDK framework changes*.

Table 6: Categorization of Android API calls

Functionality	**Number of APIs**	**Example**
Power Management	3135	WakeLock.acquire()
Local Area Wireless Networks	2116	WifiLock.acquire()
Telecomm Networks	1691	SmsManager.sendTextMessage()
Haptic Feedback	783	Vibrator.vibrate()
GPS	146	LocationManager.requestLocationUpdates()
Audio/Video	94	Camera.startPreview()
Storage	66	DownloadManager.enqueue()
Others	25	SensorManager.getAltitude()

We instrument the application code locations which invoke any API calls from our constructed pool. This instrumented code runs in an emulator on our desktop PC. The sole intention of this instrumentation is to collect the API call traces during the execution of an *event trace*. We execute the instrumented code on the emulator and record the API calls invoked for each *event trace*. These API calls are annotated with the EFG node corresponding to the triggered event. Thus, for each event trace generated from the EFG, we can generate the respective *API call trace*. It is important to note that the *event traces* are executed on the smartphone, as well as in the emulator. *The instrumented application runs on the emulator whereas the instrumentation-free application run on smartphone.* Therefore, the instrumentation does not influence the energy consumption behaviour of the application.

5.4.2 Test Generation

In this subsection, we shall describe (i) technique for hotspot/bug detection (ii) guidance heuristic for the framework and (iii) algorithm for test-generation

(i) ***Technique of Hotspot/Bug Detection***: As described in section 5.3, energy hotspots/bugs are those regions of code that lead to high E/U ratio (*cf.* Def 5.3.2). To detect energy hotspots during an event trace T, we must first obtain the E/U ratio trace (E/U_T), during the execution of T. E/U_T is divided into four different stages: pre-execution stage (*PRE*), execution stage (*EXC*), recovery stage (*REC*) and post execution stage (*POST*) (see Figure 27). The rationale for dividing E/U_T trace into four stages is as follows: in the *PRE* stage the execution of event trace T has not started yet. Therefore, *PRE* stage records the idle-behaviour (low-power state) of the device. Similarly, in the *POST* stage, the devices has completed execution of T and so in an ideal scenario the device would have gone back to its idle-behaviour during *POST* stage. The execution stage, as the name suggests, is when T is actually executing on the device. After the execution of T, the device takes a brief period of time (referred to as *screen-time-out* duration) to return to its idle-behaviour. In our framework this time period between the *EXC* and *POST* stage is referred to as *REC* stage [2].

To detect the presence (or absence) of an energy bug we compare the E/U_T values in *PRE* and *POST* stages using statistical methods. If the dissimilarity between E/U_T values in *PRE* and *POST* stage is more than a predefined threshold (in our experiments the threshold was set to 50%), an energy bug is flagged (*i.e* execution of T changed the idle-behaviour of the device).

Compared to detection of bugs, detection of hotspots is much trickier. Hotspots may appear only during the execution of an event trace (*i.e. EXC* stage) or just after the execution of an event trace (*i.e. REC* stage) stage. Note that E/U_T values obtained in *EXC* stage and *REC* stage may substantially vary for different event traces. Besides, different executions of the same event trace may show different E/U_T values in *EXC* stage or *REC* stage, due to different hardware states. Therefore, we first need a clear definition of energy hotspots to detect them automatically. We believe that *abnormally high energy wastage during the execution of an event trace* is a suitable indicator of energy hotspots. To detect such unusual energy behaviours, we draw connections from the data mining and classification techniques. We observe that the problem of detecting unusual energy behaviours is similar to detect *unusual subsequences in time-series data*. We use an anomaly detection technique that computes *discords* [102] in a time-series data. *Discords* are subsequences in a time-series data, that

2 In all our experiments, REC stage was much larger than the *screen-time-out* duration. This allowed the device to return back to its idle behaviour by the *POST* stage after a bug-free event trace has completed execution.

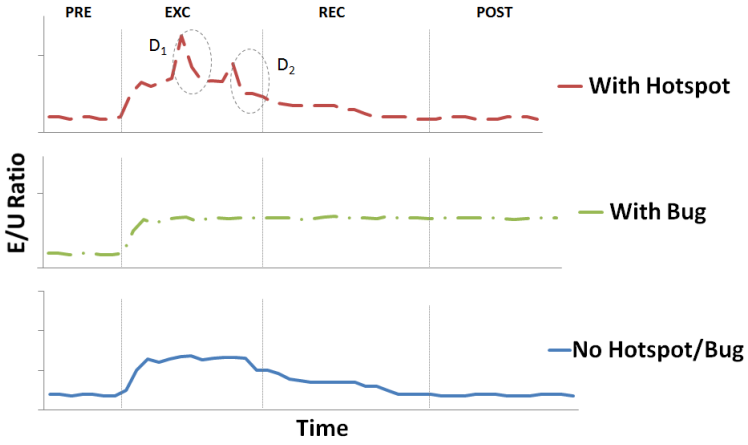


Figure 27: An example of energy-consumption to utilization (E/U) trace with no hotspot/bug, with an energy bug and with an energy hotspot

are maximally different from the rest of the time-series. We employ the *discord* detector on the E/U_T values from the *EXC* and *REC* stage. As a result, the discord detector highlights subsequences in E/U_T that are abnormally different from the rest of the subsequences in the *EXC* and *REC* stage. Additionally, the anomaly detector also points out the magnitude of each computed discord. For instance, in Figure 27, discord D_1 has a higher anomaly magnitude than the discord D_2. These magnitudes are extremely helpful. This is because the computed energy hotspots can be *ranked* based on their magnitude, before reporting to the developers. As the anomaly detector, we integrate `JMotif` [98] into our framework. `JMotif` is an off-the-shelf data mining library and it includes the implementation of finding discords in a time-series data, as proposed in [102].

(ii) ***Guidance Heuristics for Test Generation***: The primary objective of the guidance heuristics is to select an unexplored event trace that has a substantial likelihood of leading to a hotspot or a bug. The guidance function uses three parameters to rank the unexplored event traces: (a) number of API calls in the event trace (b) similarity to previously explored, hotspot/bug revealing event traces (c) starvation of event traces due to unexplored API calls. The rationale for using these parameters is explained subsequently.

We have described in an earlier section (4.1:(iii) Extraction of API calls) that the major power consuming component in smartphones can be accessed through a set of API calls. Therefore, the presence of API calls that activate (or deactivate) such hardware components can be used for guiding our test generation. At the beginning of test generation process, all event traces are ranked according to the number of such API calls they can invoke. In subsequent iterations, the guidance module becomes more intelligent by learning specific API call sub-sequences that are more likely to generate energy hotspots/bugs, which is where the guidance by similarity (or exploration history) comes into play. While selecting an unexplored event trace, the guidance heuristics compares an unexplored trace to all previously explored event traces that had uncovered an energy hotspot or a bug. Comparison between two event traces is performed in terms of the sequence of API calls they can invoke. Note that such a comparison is perfectly *feasible*, as we extracted the API call trace for each event trace during the preprocessing stage. Similarity between two API call traces is compared using *Jaro Winkler Distance* algorithm [103]. Finally, our third parameter, guidance by starvation, aims to cover as many API calls as possible during exploration. Since the first two parameters are

based on the number of API calls and the exploration history it is possible that the guidance heuristics may ignore several unexplored API calls. This leads to *starvation*, where a set of API calls will never be explored by the test generation process. Such starvation is undesirable, as unexplored API calls may potentially expose new energy hotspots/bugs. Therefore, to ensure a fair coverage of all the API calls invoked by an application, we add a guidance parameter to deal with the problem of starvation. Essentially, guidance by starvation ranks all unexplored event traces by the ratio of number of unexplored API calls in an event trace to the total number of API calls in all event traces.

(iii) ***Algorithm for Test-Generation***: The algorithm for our test-generation framework is shown using a flow chart (see Figure 28). The primary objective of our framework is to uncover as many energy hotspots/bugs as possible in an application, within a given time budget. Input to our framework is an Android application from which the database of the application's event traces is generated. Recall that generation of event traces from the EFG of an applications was explained in section 5.4.1. Our framework systematically executes the event traces from the database on the smartphone. Each execution is monitored for presence of hotspots/bugs. The exploration continues until the allocated time budget has expired. On completion, the framework reports a set of event traces, each of which leads to an energy hotspot/bug when executed on the device. The two most important components of our framework, that are *Guidance heuristics for test generation* and *Technique of hotspot/bug detection*, have been discussed in preceding paragraphs. There is however, one more component of the framework that must be explained. Notice that in the flow chart (Figure 28), the first block indicates *Refine Guidance Parameters,* α, β, γ. Essentially, this indicates the step in our framework where the reliance (or the weight) of the various guidance parameters are refined. Recall that our guidance heuristics is based on three parameters, guidance by number of API calls (corresponding weight would be α), guidance by exploration history (corresponding weight would be β) and guidance by starvation of API calls (corresponding weight would be γ). Assume that, for a given event trace E, guidance by number of API calls assigns a rank G_n, similarly guidance by exploration history assigns a rank G_h and guidance by starvation assigns a rank G_s. To obtain a single score S_E for an unexplored event trace E, we use equation 33.

$$S_E = \alpha \times G_n + \beta \times G_h + \gamma \times G_s \tag{33}$$

where $\alpha + \beta + \gamma = 1$. In equation 33, α, β and γ are three tunable factors which drive the priorities of different guidance parameters. In the beginning, we do not have any knowledge about likely hotspots/bugs. Therefore, α is initialized to 1 and both β and γ are initialized to 0. In each iteration, the value of α, β and γ are refined to uncover likely energy hotspots/bugs, as well as to get a fair coverage of invoked API calls. Specifically, in each iteration, we decrease the value of α by a fixed amount Δ ($0 < \Delta < 1$). If an energy hotspot was found in the previous iteration, we increase the value of β to $\beta + \Delta$. The intuition behind this refinement is to find energy hotspots/bugs that had similar API call sub-sequences as previously found hotspots/bugs. We continue increasing the value of β as long as we find hotspots/bugs or the value of β reaches 1. If we are unable to find any hotspots/bugs in some iteration, we hope to reach previously unexplored API calls and therefore, we increase the weight of γ to $\gamma + \Delta$. This assignment of extra weight Δ is taken out from α, if $\alpha \geq \Delta$. Otherwise, we modify the value of β to $\beta - \Delta$ to decrease the priority of execution history.

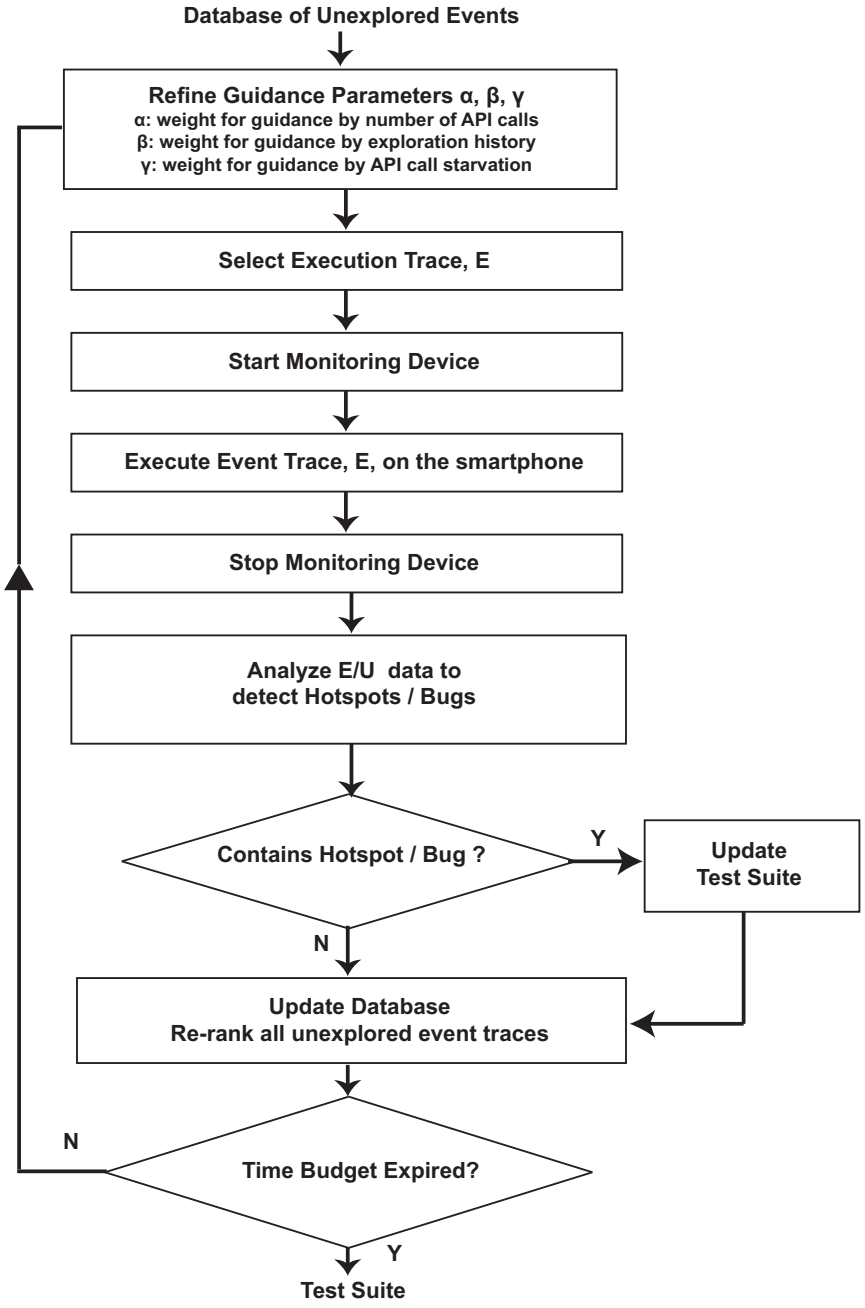


Figure 28: Flow chart for our test-generation framework

Table 7: Statistics for all the Energy Hotspots/Bugs found in tested applications (out of the 30 applications that we analyzed)

Application	Description	Feasible Traces (k=4)	Bugs Found / False Positive	Hotspots Found / False Positive	Hotspot / Bug Type	Previously Reported
Aagtl	A geocaching tool	131	Yes / No	Yes / Yes	Resource Leak	No
Aripuca	Records tracks and waypoints	502	Yes / No	No / n/a	Vacuous Background Services	No
Montreal Transit	Fetches bus, subway and other transit information	64	No / n/a	Yes / no	Expensive Background Services, Suboptimal resources binding	No
Omnidroid	Automated event/action manager	233	Yes / No	No / n/a	Vacuous Background Services, Immortality bug	Yes
Zamnim	Shows location-aware zmanim	965	Yes / No	No / n/a	Vacuous Background Services	Yes
Sensor Test	Monitors and logs sensor output	2,800	Yes / No	No / n/a	Immortality bug	No
Eponte	Displays traffic information	200	No / n/a	Yes / No	Suboptimal resources binding	No
760 KFMB AM	Listens to online radio	26	Yes / No	Yes / No	Vacuous Background Services, Suboptimal resources binding	No
Food Court	Finds restaurants near a location	42	Yes / No	No / n/a	Vacuous Background Services	No
Fire and Blood	Simple touch and draw game	156	Yes / No	No / n/a	Vacuous Background Services	No
Speedometer	Shows measurements of sensors	2,492	Yes / No	No / n/a	Vacuous Background Services	No

5.5 EXPERIMENTAL EVALUATION

We evaluated our framework to answer the following three research questions: *(i)* Efficacy of our framework in uncovering energy bugs and hotspots in real-world applications, *(ii)* How can an application developer benefit from the reports generated by our framework, and *(iii)* Is guidance based on API call coverage more appropriate metric than code coverage for uncovering energy bugs and hotspots? First, we describe our experimental setup and the set of subject programs that we analysed in our experiments.

5.5.1 Experimental Setup

In our experiments, we use an LG Optimus L3 smartphone as the device to run our subject programs. The device has a single core processor and features standard I/O components such as GPS, WiFi, 3G and Bluetooth. The device uses Android 2.3.3 (Gingerbread) operating system (OS). To monitor energy consumption of the smartphone, we used a Yokogawa WT210[104] digital power meter for precise power measurement. Our energy-testing framework runs on top of a Desktop-pc that has an Intel Core i5 processor and 4 GB RAM. The OS used on our Desktop-pc was Windows 7.

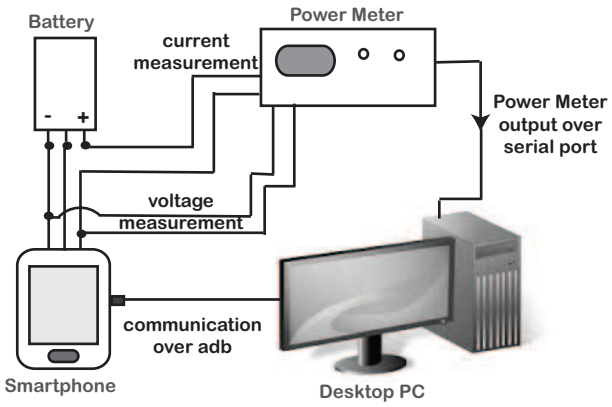


Figure 29: Our experimental setup

Figure 29 shows the setup for our experiments. For the purpose of this experiment we created a special apparatus to house the smartphone battery, such that we could measure the voltage and current flowing through the battery without any distortion. Note that contemporary smartphone batteries may have more than two terminals. Additional terminals may be used by the battery to report data such as internal temperature. However, for our experiments only the positive and the negative terminals need to be monitored (as shown in Figure 29). Any additional terminals may be directly connected to the smartphone. Our framework runs on the Desktop-PC, which also serves as the global clock. All the measurements from the power-meter (reporting power consumption data) and the smartphone (reporting utilization data) are collected at the Desktop-PC. Each reading is recorded with a timestamp generated on the Desktop-PC. Since the timestamps are generated by a single clock (the clock on the Desktop-PC) we can use these timestamps to synchronize [105] the data from the power-meter and the smartphone. Also note that we use the *android debug bridge* to communicate with the smartphone. These communication includes sending event traces to the smartphone and recording utilization data.

5.5.2 Choice of Subject Programs

The subject programs for our experiments are available on Google Play store/F-droid repository [106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135]. We have analyzed a total of 30 Android applications from different categories (*e.g.*tools, productivity, transportation) as shown in Figure 30. The subject programs are diverse in terms of *apk* (Android application package file) size. The largest application tested was 8.0MB in size while the smallest application was 22KB in size. The average *apk* size of the subject programs was 1.1MB. The subject programs also had varying GUI complexity. We measure GUI complexity of an application by the number of feasible event traces that could be explored, starting from the main screen of the application. By fixing the length of the event traces to explore (to a length of 4), we observe that our chosen subject programs contain between 26 to 2,800 feasible event traces. We also estimate the popularity of an application by observing the number of times it has been downloaded, as well as its user ratings. These two statistics are only available for applications on the Google Play store. As of March 10, 2014, the subject programs have an average user rating of 4.0 out 5, with a minimum rating of 2.7 and a maximum rating of 4.6. The median number of downloads for the subject programs is between 10,000 - 50,000, with a minimum download count of 1,000 and a maximum download count of 10,000,000.

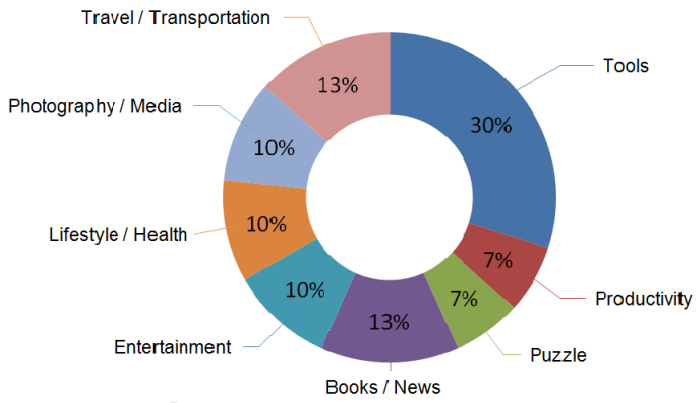


Figure 30: Categories of the 30 Android applications used in our experiments

Note that our framework does not require the source code of an application to detect energy hotspots/bugs. However, the source code is required to obtain code coverage metrics and for debugging purposes. Therefore, we only consider open-source applications for the second and third research questions (where the source code is needed to perform our evaluation). The lines of code for the open-source applications used in our experiments varied from 448 to 11,612, with an average of 4010 lines of code per application.

5.5.3 Results

RQ1: Efficacy of our framework in uncovering energy bugs and hotspots in real-world applications

One of the objective of our experiments was to observe the efficacy of our framework to quickly uncover energy bugs and energy hotspots in real world applications. To do so, we evaluated the applications using our framework with a time budget of 20 minutes. Additionally, we also limit our exploration for event traces up to a length of 4. A summary of the bugs and hotspots reported by our framework is listed in Table 7.

Our framework reported energy bugs for 10 out of 30 subject programs. The framework also reported energy hotspots for 3 subject programs. Note that our hotspot detection technique is based on an anomaly detection method [102]. Therefore, some of the reported hotspots may contain *false positives* due to the presence of noise in the measured data. Such noise may arise due to unpredictable behaviours such as network load. To confirm a reported energy hotspot, we manually execute the respective event trace on our device and we observe whether the same energy hotspot can be replicated. The result of the manual validation (*cf.* Table 7) revealed only one *false positive*, for the application *Aagtl*. It is important to note that the number of *feasible* event traces can be substantially large even for event traces having length 4 (as shown in Table 7). In spite of this large number of event traces, we observed that our framework can quickly gravitate the exploration process towards more energy-consuming event traces. Existing tools for Android application UI testing , such as Monkey, cannot uncover such high energy consuming event traces because they are designed to stress test the UI of the application by generating pseudo random stream of user events irrespective of the application's EFG or the API call usage.

RQ2: How can an application developer benefit from the reports generated by our framework?

After analyzing an application, our framework generates a *test report*. This report serves as a guide to optimize energy consumption and to remove potential energy issues. The report contains a set of test cases, where each test case captures an energy issue reported by our framework. Each test case includes *(i)* a *MonkeyRunner* script for automatic execution of events that lead to the energy issue , *(ii)* energy trace pattern *(iii)* details of the energy issue (*e.g.* magnitude of energy hotspot) and *(iv)* the set of API calls invoked.

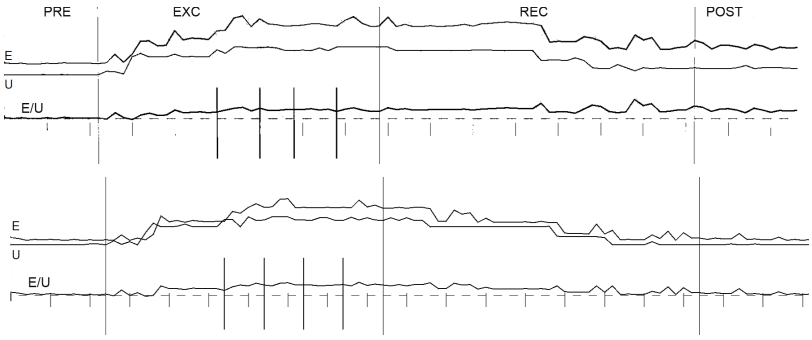
From the report, the developer may prioritize energy issues exhibiting an energy bug or an energy hotspot of relatively high magnitude. For each test case, the developer can run the provided *MonkeyRunner* script and observe the event sequence that navigates the application to trigger the reported energy issue. This would help the developer in identifying the root cause of the energy issue. For instance, let us assume that an event trace T exposes an energy hotspot. While executing T, if the hotspot appears before the execution of a certain event E, neither E nor any subsequent events in T are responsible for causing the hotspot. Thus, the search space for identifying the root cause of the hotspot is reduced to the code fragments that were executed before E was triggered. This will help the developer in fixing the reported energy issues. We have performed case studies on two of the analyzed applications (our framework reported energy bug for one and a hotspot for another) to demonstrate how a developer can utilize the generated reports to debug and fix energy issues in applications.

ARIPUCA GPS TRACKER Our framework reports two event traces with energy bugs in *Aripuca GPS Tracker*. The energy consumption pattern for such an event trace is shown in Figure 37(a). As shown in Figure 37(a), the energy consumption in the *POST* stage is not similar to the *PRE* stage, indicating an energy bug. Therefore, the device did not become *idle* even in the absence of user activity. The effect of the bug is permanent, unless *(i)* GPS location update is explicitly removed, or *(ii)* the application is killed. We manually verified that the reported event traces do not exercise the functionality of the application that requires GPS location update to run in the background. The reported event traces were:

`waypointsButton` – `waypoint_details` – `MenuButton` – `button1`

`waypointsButton` – `waypoint_details` – `MenuButton` – `BackBtn`

The reported bug indicated that the location updates (GPS updates) were not removed before the application becomes inactive. By observing the similarity between the two traces (*i.e.* the event sequence `waypointsButton` – `waypoint_details` – `MenuButton`), we deduced that the bug was triggered upon arriving at a certain GUI state. We manually execute the event trace `waypointsButton` – `waypoint_details` – `MenuButton` and suspend the application afterwards by pressing the *Home* button. At this point, location updates are not needed by the application any more and they should be switched off. Upon inspecting the source code, we observed that the application had a missing code fragment for removing location updates when exiting. We fix the issue by adding the release code at an appropriate location. Thereafter, we re-run the reported event trace using our framework. The energy consumption graph after fixing is shown in Figure 37(b). As shown in Figure 37(b), the energy consumption in the *POST* stage is similar to the *PRE* stage, resolving the energy bug.



(a) Before fix

(b) After fix

Figure 31: Energy trace of the event trace for Aripuca GPS Tracker

MONTREAL TRANSIT Five event traces with hotspots are reported for the application *Montreal Transit*. The energy consumption trace for one such event trace is shown in Figure 32(a). Immediately after the execution enters the *REC* stage (*cf.* Figure 32(a)), we can observe potentially high E/U ratios in a period of around 5 seconds. Note that this energy issue is an energy hotspot and not an energy bug. This is because the high E/U ratio does not persist. The code for pausing the application consumes abnormally high amount of energy, causing the hotspot to appear during the same period. We observed that all the five reported event traces exhibit similar hotspots. On a closer inspection, we found that the GPS location updates continue to run for a few seconds even after the application exits. Before we explain the exact cause for the hotspot, let us first give an overview of the application.

Montreal Transit is an application to show transit information, where each screen shows transit information for some mode of transportation. When a screen for some transportation, say subway, is displayed, it fetches the distances to some of the nearest subway stations. However, in order to do so, it needs to acquire the location of the device. Surprisingly, we found that the location update was triggered twice, instead of once. The second location update was triggered by a *third-party advertisement module* to display location-based advertisements in the application. We found that the code to load advertisement is being executed on the main thread of the application. As a result, any delay in loading the advertisement from the network prolongs the entire main thread. If the user exits the application while the main thread is being

delayed, the release of GPS based location updates is delayed as well. The hotspots reported in our experiment can be best explained by such delay. To confirm our speculation, we moved the code related to the loading of advertisements in a separate asynchronous thread. As a result, we observed that the event traces which earlier exhibit hotspots, no longer do so (*cf.* Figure 32(b)). On a different note, we suggest that to develop energy-efficient applications, the developer should use expensive resources as optimally as possible. For instance, the location updates in the preceding scenario should be performed just once and shared between the various modules that need it. We also suggest that any feature that is surplus to the requirements of users (*e.g.* advertisements), should be put in a separate asynchronous thread to improve the user experience.

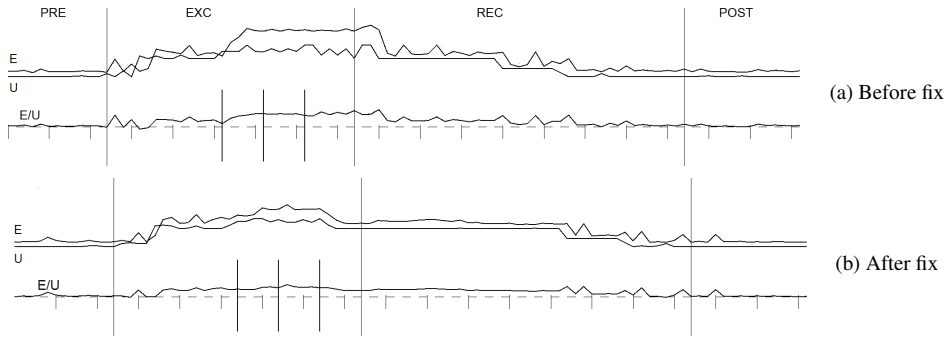


Figure 32: Energy trace of the event trace for Montreal Transit

RQ3: Is guidance based on API call coverage more appropriate than code coverage for uncovering energy bugs and hotspots?

We have argued that I/O components and power management utilities contribute significantly to the energy consumption of a mobile device. Therefore, we use the number of API calls invoked by an event trace as one of the guiding parameters in the exploration. As a result, the test-suite generated by our framework should cover as many API calls as possible. On the other hand, a more conventional approach would be to measure code coverage of the test-suite to evaluate the efficacy of a test-generation framework. Therefore, we evaluated the efficacy of API call based coverage with respect to code coverage, to obtain a minimal test suite for uncovering energy bugs or energy hotspots.

We choose *line of code* (LoC) as our code coverage metric and use *EMMA*, a Java code coverage tool, to obtain LoC covered by a test suite compared to the total LoC of the application. We observed that the generated test-suites had a API call coverage of more than 83%, while having code coverage ranging from $11\% - 90\%$ (see second and third columns of Table 8). Subsequently, we wanted to observe if achieving an incremental code coverage uncovers any additional hotspots/bugs. Therefore, we manually generated additional test cases for the applications in Table 8. We observed that the manually generated test cases increased the code coverage ranging from 4% to 17%. However, no additional hotspots/bugs were revealed. This is most likely due to the inefficiency of a human user to systematically find energy-inefficient event traces, based on a given metric. Additionally, on inspecting the *EMMA* coverage reports (and the source code), we observed that for real-life applications, a substantial portion of the code is present to give feedback to the user and to ensure compatibility over different versions of the OS. Therefore, the coverage achieved by executing such code would not necessarily contribute to finding energy hotspots/bugs.

Table 8: Coverage statistics from all open-source apps used in our experiments

Application Name	API Call Coverage (%)	Code Coverage (%)	Lines of Code
Aagtl	100	21	11,612
Android Battery Dog	100	17	463
Aripuca	100	15	4,353
Kitchen Timer	100	30	1,101
Montreal Transit	89	11	10,925
NPR News	100	24	6,513
Omnidroid	83	36	6,130
Pedometer	100	56	849
Vanilla Music Player	86	20	4,081
Simple Chess Clock	100	49	448
WiFi ACE	100	27	504
World Clock	100	90	1,147

5.6 COMPARISON WITH EXISTING TECHNIQUES

In recent times, due to the prevalent use of smartphone devices, topics related to functional and extra-functional testing of smartphone applications have attracted the attention of software engineering research community. Recent proposals, such as [82] and [81], have discussed functionality testing of Android applications based on symbolic execution and biased random search. In contrast, we focus on automated testing of extra-functional aspects for smartphone applications, specifically the energy behaviour.

Recent works on energy-aware profiling [3, 72] have shown poor energy behaviour of several smartphone applications. These works on profiling validates the idea of energy-aware development for smartphone applications. However, like any other program profiling techniques, works proposed in [3, 72] require specific input scenarios to execute the application on smartphone device. A more recent work [74] has proposed a technique to relate power measurements with source lines of applications. Such a technique also requires input scenarios to execute an application. Automatically finding such input scenarios is extremely non-trivial, as the poor energy behaviour might be exposed only for a specific set of user interaction scenarios. Therefore, our approach on generation of input scenarios complements the works proposed on energy-aware profiling or source-line level energy estimation. Once the set of user interaction sequences is generated by our framework, they can be further used with works such as [3, 72] or [74].

The work proposed in [79] discusses energy-aware programming support via symbolic execution. For each code path explored by a symbolic execution toolkit, the base energy cost can be highlighted to the programmer. However, such an approach is preliminary in the sense that it only considers the CPU power consumption. In contrast, power consumption to access memory subsystems, network card and other I/O components were not considered. In smartphone devices, I/O components consume the most power. Since we perform direct power measurements for an application, we can highlight the gross energy consumption to the developer, without ignoring the energy consumption of any hardware component. The work in [73] proposes to analyze the overall energy behaviour of an application via an energy model. Our goal is orthogonal to such approach. We aim to find user interaction scenarios that may lead to undesirable energy behaviours of an application. Therefore, our work has a significant testing flavour compared to the work proposed in [73]. More importantly, we

rely on direct power measurements rather than relying on any energy model. Another work [76] uses *data flow analysis* to detect *wakelock* bugs in Android applications. The detection of wakelock bugs is relatively easy. This is due to the fact that the acquire and release of wakelocks can be related directly to program statements. Therefore, the detection of wakelock bugs can be performed even in the absence of power measurements. In contrast, we aim to solve a more general problem of detecting energy inefficiencies and in addition, we also compute the specific input scenarios that witness the same.

A different line of work aims to produce energy-efficient applications from different implementations of the same functionality [84, 85]. The decision to choose an implementation is influenced by monitoring the power consumption for a *given test-suite*. For instance, the work in [84] dynamically chooses approximate implementations of a given functionality to reduce the power consumption. Along the same line, a recent work [85] monitors the power consumption of different API implementations and computes the potentially best implementation in terms of energy-efficiency. Our work is complementary to such approaches, as we aim to automatically detect input scenarios that result in energy inefficiencies and generate a test-suite that can be used for improving the energy-efficiency of the application. Finally, the work in [86] introduces programming language constructs to annotate energy information in the source code. Since we directly measure the power consumption, our approach does not require any new language construct.

5.7 CHAPTER SUMMARY

In this chapter, we provide a systematic definition, detection and exploration of energy hotspots/bugs in smartphone applications. Our methodology is used to develop a test-generation framework that targets Android applications. Each entry in our generated test report contains a sequence of user-interactions that leads to a substantial wastage of battery power. Such test cases are useful to understand several corner scenarios in an application, in terms of energy consumption. Our evaluation with 30 applications from Google Play/F-Droid suggest that our framework can quickly uncover potential energy hotspots/bugs in real-life applications. It is worthwhile to mention that our test generation method is not *complete*. This is due to the fact that our computed event flow graph (EFG) may only cover a portion of the application. As a result, we may not expose *all the energy hotspots/bugs* in an application. Besides, our current test generation framework revolves around directing the test generation towards I/O operations, as I/O components are some of the major sources of energy consumption in smartphones. However, it is possible in some pathological cases (*e.g.* unusual cache thrashing and memory traffic) that CPU-bound applications may lead to substantial drainage of battery power. Detection of such energy stressing behaviours can be studied in the future. In our current implementation, we can deal with GUI-based applications by generating UI inputs automatically. However, certain applications (such as *game applications*) require human intelligence in navigating through the different GUI screens. For example, the transition between two GUI screens might happen only by answering questions that require human intelligence. In such a situation, we may not be able to generate sufficient event traces automatically to stress the energy behaviour.

6 REPAIRING RESOURCE LEAKS TO IMPROVE ENERGY-EFFICIENCY OF MOBILE APPS

Increased usage of mobile devices, such as smartphones and tablets, has led to widespread popularity and usage of mobile apps. If not carefully developed, such apps may demonstrate energy-inefficient behaviour, where one or more energy-intensive hardware components (such as Wifi, GPS, *etc*) are left in a high-power state, even when no apps are using these components. We refer to such kind of energy-inefficiencies as energy bugs. Executing an app with an energy bug causes the mobile device to exhibit poor energy behaviour and drastically shortened battery life. Since mobiles apps can have huge input domains, therefore exhaustive exploration is often impractical. We believe that there is a need for a framework that can systematically detect and fix energy bugs in mobile apps, in a scalable fashion. To address this need, we have developed EnergyPatch, a framework that uses a combination of static and dynamic analysis techniques to detect, validate and repair energy bugs in Android apps. The use of light-weight, static analysis techniques enables EnergyPatch to quickly narrow down the potential program paths along which energy bugs may occur. Subsequent exploration of these potentially-buggy program paths using dynamic analysis technique helps in validations of the reported bugs and to generate test cases. Finally, EnergyPatch generates repair expression to fix the validated energy bugs. Evaluation with real-life, apps from Google Play/F-droid store shows that EnergyPatch is scalable and can produce results in reasonable amount of time. Additionally, we observed that the repair expressions generated by EnergyPatch could bring down the energy consumption on tested apps between 10% to 60%.

6.1 INTRODUCTION

Over the recent years there has been an increased usage of complex applications on battery powered mobile devices, such as smartphones and tablets. Such mobile applications or apps exploit a wide variety of sensors and other hardware components available on modern smartphones to provide a diverse set of functionalities. There is however one factor which greatly limits the usage of such apps. Battery power on mobile devices is often a constrained resource. Therefore, it is worthwhile to test and remove energy-inefficiency in such mobile apps before deployment.

In this chapter, we shall present our framework (and tool) EnergyPatch, that can help app-developers to detect, validate and repair a specific class of energy-inefficiencies in mobile apps, which we refer to as energy bugs. Executing an application containing an energy bug may cause the mobile device to consume excessive amounts of battery power even after the buggy application has completed execution and there is no user-activity. Such excessive energy consumption can drastically reduce the battery life of the mobile device, in a relatively short period of time[1]. In our recent work [2], we have observed that inappropriate usage of energy-intensive, hardware components (such as Wifi, GPS) or power management utilities (such as Android Wakelocks) may give rise to energy bugs. We also observed that such hardware components/power management utilities can only be accessed by an app through a predefined set of API calls. These observations indicate that inappropriate usage of API calls that give access to such hardware resources/power management utilities leads to energy bugs, resulting in energy-inefficient apps and shortened battery life. Hence, there is a need for a

framework that can provide an end-to-end solution to address the challenges associated with detection, validation and repair of energy-inefficiencies related to energy bugs. In particular, such a framework should be able to address the following questions:

i. How to determine if an app contains an energy bug, in a scalable fashion ?

ii. How to generate test-cases that can demonstrate the presence of energy bugs, in an automated fashion ?

ii. How to generate repair expressions that can fix the reported energy bugs ?

iv. How to bring this (detection — test generation — repair) functionality to commonly used mobile-app development platforms such as Eclipse ADT ?

Our framework, EnergyPatch, is the culmination of our effort to answer these questions. EnergyPatch extracts a model of the app (under test), using automated analysis. It then analyses this model using a light-weight, static analysis technique to detect program paths that may potentially lead to an energy bug. These potentially-buggy program paths are then explored using a dynamic analysis technique to validate the presence of energy bugs. During exploration, if the presence of an energy bug is validated, EnergyPatch generates test-cases that bear witness to the presence of the reported energy bug. Finally, EnergyPatch, generates repair expressions for the reported energy bugs.

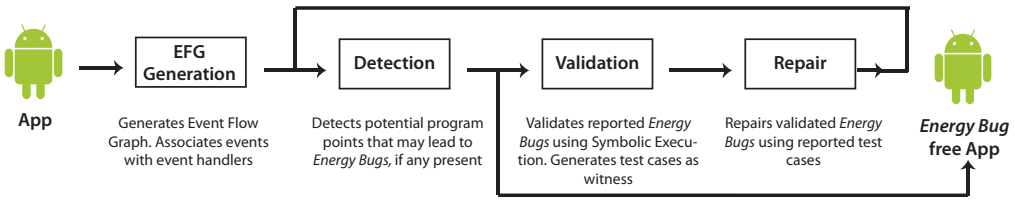


Figure 33: System overview

Figure 33 shows the three key phases EnergyPatch: detection, validation and repair. More specifically, the detection phase is based on abstract interpretation technique, the validation phase is based on symbolic execution technique and the repair phase uses template based repair. It is worthwhile to know that using symbolic execution alone to explore non-trivial programs may lead to the problem of path-explosion. To ensure scalability of framework, we use a multi-staged approach that is described in the following paragraphs.

To begin with, in the detection phase, our framework conservatively computes the set of program paths along which energy bugs may occur. If no such program paths can be found through detection phase, we conclude that no energy bugs are indeed present in the app. It is worthwhile to know that results of the detection phase are always *sound* (*i.e.* if there is an energy bug in the tested app the detection phase will always report it). On the other hand, if detection phase reports presence of program paths with potential energy bugs, we proceed to validation phase. However, before executing the potentially-buggy program paths using symbolic execution, we employ a couple of search-space reduction techniques to reduce the time required for exploration. These search-space reduction techniques are based on transitive closure computation and program slicing. Transitive closure computation (conservatively) determines which event (and corresponding event-handlers) in the app will never occur on potentially-buggy program paths (as reported by the detection phase). Program slicing is used

1 One of the buggy applications we evaluated - Sensor Tester, drained a fully charged battery on a LG Optimus E400 smartphone in less than 8 hours whereas the standby time for the phone is approximately 600 hours [136]

to (conservatively) estimate the subset of program inputs that do not influence the execution of the potentially-buggy program paths. Finally, in the repair stage we use the information provided by the previous phases to generate repair expressions for validated energy bugs.

We have implemented our framework as an Eclipse Plugin, named EnergyPatch. It is freely available from BitBucket[137]. Since a large number of Android app developers use Eclipse ADT Toolset for Android app development, we believe that our framework will be most useful in this form. Additionally, our tool provides an intuitive user interface that helps the developer in visualizing the potential energy bugs in the app. For the evaluation of our framework and tool we created a test suite using thirty five real-life Android apps. Our framework was able to detect energy bugs for twelve out of these thirty five tested apps. The test cases generated by our framework were manually executed on a mobile device and resultant power consumption measured by a power meter to confirm the presence of energy bugs. On measuring the energy consumption of the buggy applications post repair we observed a reduction in energy consumption between $10\% - 60\%$. Finally, we conclude the evaluation of our framework by comparing it with existing research works on detection and/or test-generation for resource leaks in mobile-apps.

6.2 ANDROID BACKGROUND

Android is a widely-used operating systems designed for mobile devices such as smartphones and tablets. Android is open-source, additionally there exist a number of freely available development tools (such as Eclipse ADT) and debugging utilities (such as Android Debug Bridge) for Android app development. All these factors motivated us to use Android apps as test subjects for our framework. In the following subsections we shall briefly describe some of the key aspects of execution model and energy-efficiency in Android apps.

6.2.1 Execution Model in Android

Android apps, in general, are composed of four key components: *Activities*, *Services*, *Broadcast Receivers* and *Content Providers*. Communication between the various components happens by means of messaging objects that are commonly referred to as *Intents*. Figure 34 shows an oversimplified representation of the execution model for Android apps.

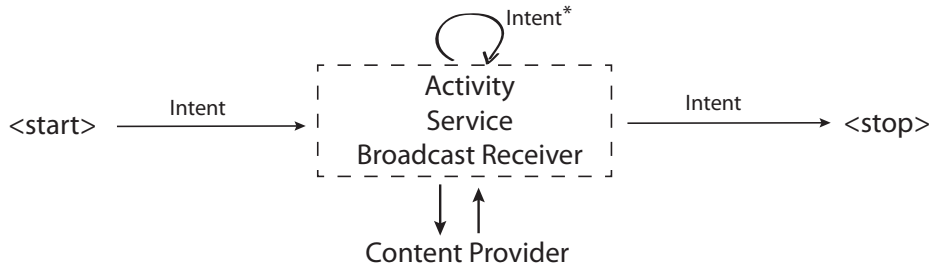


Figure 34: Over-simplified representation of execution in Android apps

An *activity* can be described as an entity which encapsulates the user interface (UI) through which the user interacts with the app. *Services*, unlike *activities*, do not have any UI associated with them. Services are run in the background and are often used for performing lengthy tasks. *Broadcast receivers*, as the name suggests, are used to receive broadcasted messages (such as arrival of call or SMS etc). *Content providers* provide access to various data sources.

All components of an app have well-defined life-cycles. For instance, the life-cycles of an *activity* is shown in Figure 21.

An *activity* goes through seven distinct stages of life-cycle throughout its execution (*cf.* Figure 21. The life-cycle stages shown in the left-arm of Figure 21 *i.e. onCreate*, *onStart* and *onResume* are invoked when an *activity* begins execution. Therefore, all tasks related to initialization and resource acquisition are usually performed in these stages. Likewise, the stages on the right-arm of the Figure 21 *i.e. onDestroy*, *onStop* and *onPause* are invoked when an *activity* stops execution. In addition, the *onRestart* stage is invoked when a previously stopped activity is restarted. To implement a custom functionality to an *activity*, app-developers simply need to override the above mentioned methods (*i.e. onCreate*, *onStart*, *onResume*, *onDestroy*, *onStop*, *onPause* and *onRestart*).

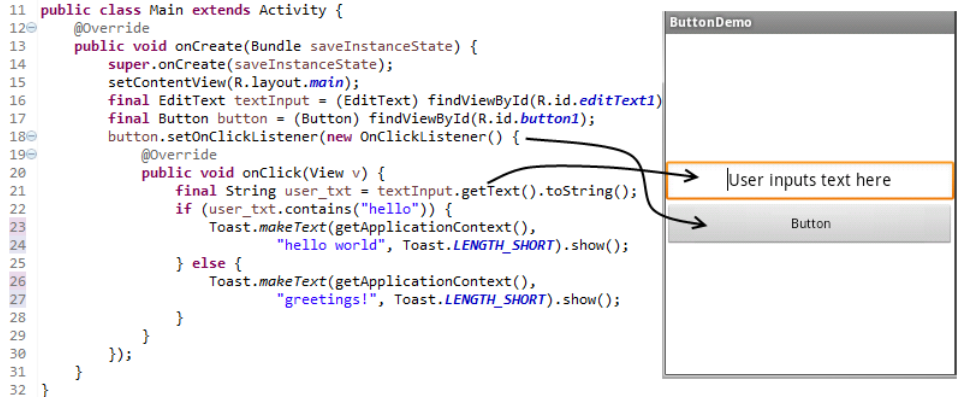


Figure 35: An example showing how inputs are provided to Android apps

6.2.2 Inputs to an Android App

As shown in Figure 36, there are two possible ways for providing inputs to Android apps *(i)* through events and *(ii)* through return value from Android API calls. Events can be user-generated (such as by pressing buttons) or system-generated (such as broadcast of change in battery state). Figure 35 shows an example of input through an event as well as through return value. In the example of Figure 35, when the user clicks the button (event), the *onClickListener* (event handler) for the button is invoked(Line 18). Subsequently, the user inputs from the text field is read (input through return value), by means of another Android API call *getText* (Line 21). Observe that the else part of the example code is executed only if the return value from *getText* does not contain the string *hello* (Line 22).

This example goes on to show that *only exploring the event of the app is not sufficient to exercise all the functionalities in the app*. Based on this understanding we define an input to an Android app as follows:

Definition 1 *Inputs to an Android app are a combination of events and return values from API calls.*

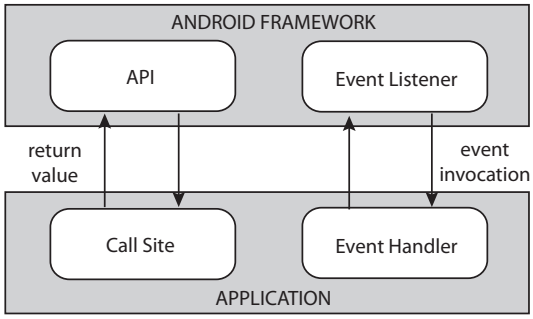


Figure 36: Inputs to an Android app

6.2.3 Energy Consumption of Android API calls

As is the case with any other non-functional property, energy consumption (or any non-functional) behaviour is seldom explicitly encoded in the application code. Therefore, in a generic scenario program analysis alone would be insufficient to determine the energy consumption behaviour of a program. However, in the specific case of Android apps we have identified a set of API calls that can substantially impact the energy consumption behaviour of the mobile device. In Android for instance, all power management utilities and I/O components must be accessed through predefined API calls. It is also worthwhile to know that these power management utilities and I/O components have significant impact on the power consumption of the device. Figure 24 shows the power profile graph of our test device *LG L3 E400*. The data for Figure 24 is obtained from the `power_profile.xml` file which is created by the original equipment manufacturer (OEM) and shipped along with the device.

We found that in a typical Android distribution, the API calls that significantly impact the energy consumption behaviour are a very small subset of all available API calls. For instance, in Android Gingerbread, which was one of the most widely used distributions at the time of writing, less than hundred of the total nine thousand or so public API calls significantly affect the energy consumption behaviour. Some of these high-energy-consuming API calls are listed in Table 9.

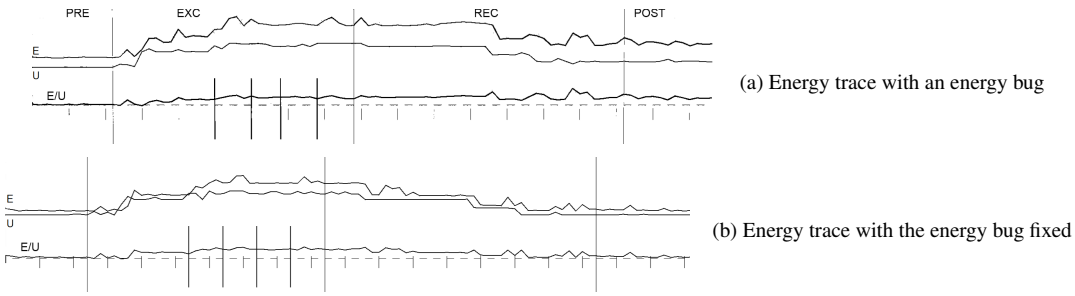
Table 9: Some of the Android API calls that have major influence on energy consumption

Resources	API calls	Hardware Resource
PowerManager (Wakelock)	acquire/release	CPU + Screen + Keypad
WifiManager	setWifiEnabled acquire/release	Wifi Hardware
Camera	open/close startPreview/stopPreview	Camera
SensorManager	registerListener/unregisterListener	Sensors
LocationManager	requestLocationUpdates/removeUpdates addProximityAlert/removeProximityAlert	GPS receiver
LocationClient	requestLocationUpdates/removeLocationUpdates addGeofences/removeGeofences	GPS receiver
MediaRecorder	start/stop	Video Hardware
AudioRecord	startRecording/stop	Audio Hardware
BluetoothAdapter	enable/disable	Bluetooth Hardware

6.2.4 Energy Bug, Cause and Effect

An energy bug is a type of non-functional defect (*i.e.* it does not affect the functionality of the app), however, it may cause the device to consume excessive power, even when no useful computation is being performed. It is worthwhile to know that energy bugs are often a manifestation of improper usage of I/O components and power management utilities. There are two possible ways in which the effect of energy-bugs can handled. These are (i) through system-level mechanisms and (ii) through testing-and-repair technique such as presented in this work. System-level techniques which aggressively release resources whenever an app exits may make the system inflexible. This is because in certain cases it is possible that the app's functionality may necessitate that resources are not released as soon as the app exits. For instance, consider a location-tracking app that wants to log the user's whereabouts throughout the day. In such a scenario, the app may want to keep working in background using GPS while the user interacts with other apps in the foreground. For such apps, forceful system-level mechanisms that aggressively release resources whenever an app exits may make the system inflexible. In comparison, a case-by-case analysis using a technique such as presented in this work can help the developer to make judicious changes to the app source-code, wherever needed.

Figure 37(a) shows the energy-consumption data from a real-world Android app, Aripuca, while an energy bug is triggered. Figure 37(b) shows the energy-consumption behaviour of the same app, for the same input but when the energy-bug has been repaired. The portion of the energy-consumption trace marked with label PRE indicates the period of time when the device is idle (no apps running). Label EXC indicates the period of time where the app is executed on the device, whereas label REC indicates the period of time the device takes to return to its idle state. Finally, the label POST indicates the period of time when the device has returned to its idle state, post execution. In an idle scenario, where there are no energy bugs involved, the energy-consumption behaviour in the PRE stage and the POST stage should be statistically similar. However, in the case of an app that has an energy bug (such as in Figure 37(a)), there will be significant dissimilarities in the energy-consumption data when comparing between the PRE and the POST stages.



(a) Energy trace with an energy bug

(b) Energy trace with the energy bug fixed

Figure 37: Energy trace for Aripuca GPS Tracker (a) with an energy bug (b) repaired energy bug. The additional energy consumption can be observed in the recovery (REC) and the post (POST) stages

6.2.5 Differences Between Present and Previous Work

The key contributions of our previous work[2] were the introduction of a fault-model for energy-inefficiencies in mobile apps and a guided, search-based test-generation framework. The energy-inefficiency detection in [2] was done using a hardware-software, hybrid approach. Like our current work, it used an automated technique [81] to generate the event-flow graph of the app (under test). However, unlike our present work, it uses a guided, search-based

algorithm to select the test-inputs to execute the app on the target mobile device. While the app is being executed on the test device (with the selected test-input) the framework simultaneously measured the power consumption using a power meter. The acquired power trace is then analysed using statistical and anomaly detection techniques to uncover energy-inefficient behaviour. In contrast, in our current work, we use a combination of a static and dynamic analysis techniques to ascertain the presence of energy bugs and to generate test-cases. The use of measurement setup (*cf.* Figure 57), in our present framework, is simply to measure the resultant energy savings. Another key difference is in the way of exploration itself. In the previous work, the exploration algorithm generates events traces, by walking through the event flow graph. However, if an UI screen needed inputs through an input-container (such as text-fields), random data was used. As a result the exploration algorithm may not have been able to explore all feasible paths inside an event-handler. In contrast, in our present work, due to the use of symbolic execution our framework can explore all feasible paths inside the event handler. To summarize, the key differences between the two work arise due to (i) the way energy bugs are detected (power measurement vs static analysis) (ii) the way test-cases are generated (search heuristics vs guided symbolic execution) and (iii.) automatic repair expressions generation (only in the current framework).

6.3 OVERVIEW BY EXAMPLE

In this section, we shall describe the workings of our framework by means of a simple example. In particular, we shall focus on (i) use of abstract interpretation to detect energy bugs and (ii) use of symbolic execution to generate test cases that leads to the reported energy bugs. In our framework, we also use an event-flow graph (EFG) generation phase as a pre-processing phase. However, for the purposes of simplicity, we shall omit the EFG generation phase in this example. We shall base our discussion on the simple code fragment shown in Figure 38(a). It has a simple input-dependent, do-while-loop, in which, a source line acquires a reference-counted resource R at Line 4 and another source line releases the acquired resource at Line 6. The only input to the program is N, which determines the number of loop iterations. Figure 38(b) shows the control flow graph (CFG) of the code shown in Figure 38(a). It can be observed that the the resource R is never released for all inputs satisfying the formula $N > 3$. Using this example, we shall first describe how static analysis is used in our framework to detect that the resource R may not be released at the end of the program (*i.e.* exit node $E4$). Subsequently, we will describe the use of dynamic analysis to generate test cases that witness the scenarios where the R hasn't been released.

6.3.1 Detection Using Abstract Interpretation

The detection phase tracks an (over)estimate of the state of resource R, at each program point. Assume that the state of resource R is denoted by a tuple $<R,k>$ at each node of the graph (cf. Figure 38(c)). The input (in) and output (out) state of resource R is shown using a tuple $<R,k>$ at each node of the graph in Figure 38(c), where $k = 0$ implies R is *not acquired* and $k = 1$ implies R is *acquired*. Every time the API call Acquire(R) is encountered the resource state is $updated$ to $<R,1>$. Similarly, whenever the API call Release(R) is encountered the resource state is $updated$ to $<R,0>$. It is worthwhile to know that resource state update operation is *object-insensitive*.

In the scenario where there are multiple incoming resource states (from different branches) we perform a $Join$ operation to merge multiple resource states into one. For instance, in the this example at the node marked $Join$, the incoming resource states from both the branches

($<R,0>$ from $i < 3$ branch and $<R,1>$ from $i \geq 3$ branch) are joined to create an over-approximated state ($<R,1>$). By performing such path-insensitive joining of states, we can avoid the problems associated with state-space explosion. The *update* and *join* operations that are described here, are applied until the *in* and the *out* states of each node in the graph do not change over an entire iteration (*i.e.* fixpoint is achieved). At the end of detection phase (once the fixpoint is reached), we check the *out* state of exit nodes to see if any resource is still in the *acquired* (*i.e.* $k = 1$) state. If so, presence of potential resource leak is reported by the detection phase. In our example (*cf.* Figure 38(c)), observe that at the exit node, the resource state is indeed $<R,1>$, indicating the presence of a resource leak. This leak will manifest for all inputs satisfying the condition $N \geq 3$. Observe that in this specific example, *if* the path satisfying the condition $i \geq 3$ had been infeasible for some reason, the results of the detection phase would have been false positive. Therefore, to rule out any possibility of false positives (due to infeasible paths/execution scenarios) we perform further analysis to validate the results of the detection phase. Essentially, we wish to validate that the property (*Prop: resource R is not acquired*) at all exit nodes. To facilitate this, our framework automatically instruments two new variables *acq_r* and *rel_r* in the program. Specifically, wherever Acquire(R) is called variable *acq_r* is incremented and wherever Release(R) is called variable *rel_r* is incremented. Additionally, at the exit node the assertion $acq\_r - rel\_r \leq 0$ is instrumented that represents the property resource R is not in the acquired state. The resultant CFG is shown in Figure 38(d).

6.3.2 Test Generation Using Symbolic Execution

The instrumented program is then explored symbolically. That is to say that the program is executed with symbolic inputs. In our example symbolic input being N. In our framework, we also perform a couple of search-space-reduction techniques to make symbolic execution phase faster. However, for the sake of simplicity we shall not describe them in this example. The objective of symbolic execution of the instrumented program is to see if any of the instrumented assertions ($acq\_r - rel\_r \leq 0$, in this example), are violated for any feasible execution. Figure 38(e) shows one possible execution path where the instrumented assertion is violated. It is worthwhile to know that if we had used bounded symbolic execution instead of unbounded symbolic execution for exploration, we may not have been able to find such a assertion violation (as demonstrated by Figure 38(f)). Assertion violation demonstrates a feasible scenario where a resource leak happens in this example. The symbolic execution also provides us with the test-cases that witness the failure of assertion. These test-cases can be used by the tester/developer to re-create the reported bugs manually.

6.4 DETECTION

In this section, we shall provide a detailed description of the detection phase. As mentioned in Section 6.3, the static analysis technique used in the detection phase is an instantiation of the abstract interpretation based approach proposed in [138]. Therefore, before describing the details of our approach we shall provide a very brief introduction to abstract interpretation itself.

To implement an abstract interpretation based program analysis framework, one needs to define the abstract semantics, in particular, (a) an *abstract domain* and (b) a set of *abstract operations*. At each program point (P) an abstract state captures the state of the program. The set of all abstract states is referred to as the abstract domain (D). The *abstract operators*, namely *update* and *Join*, can be used to manipulate the abstract state to reflect the effect of

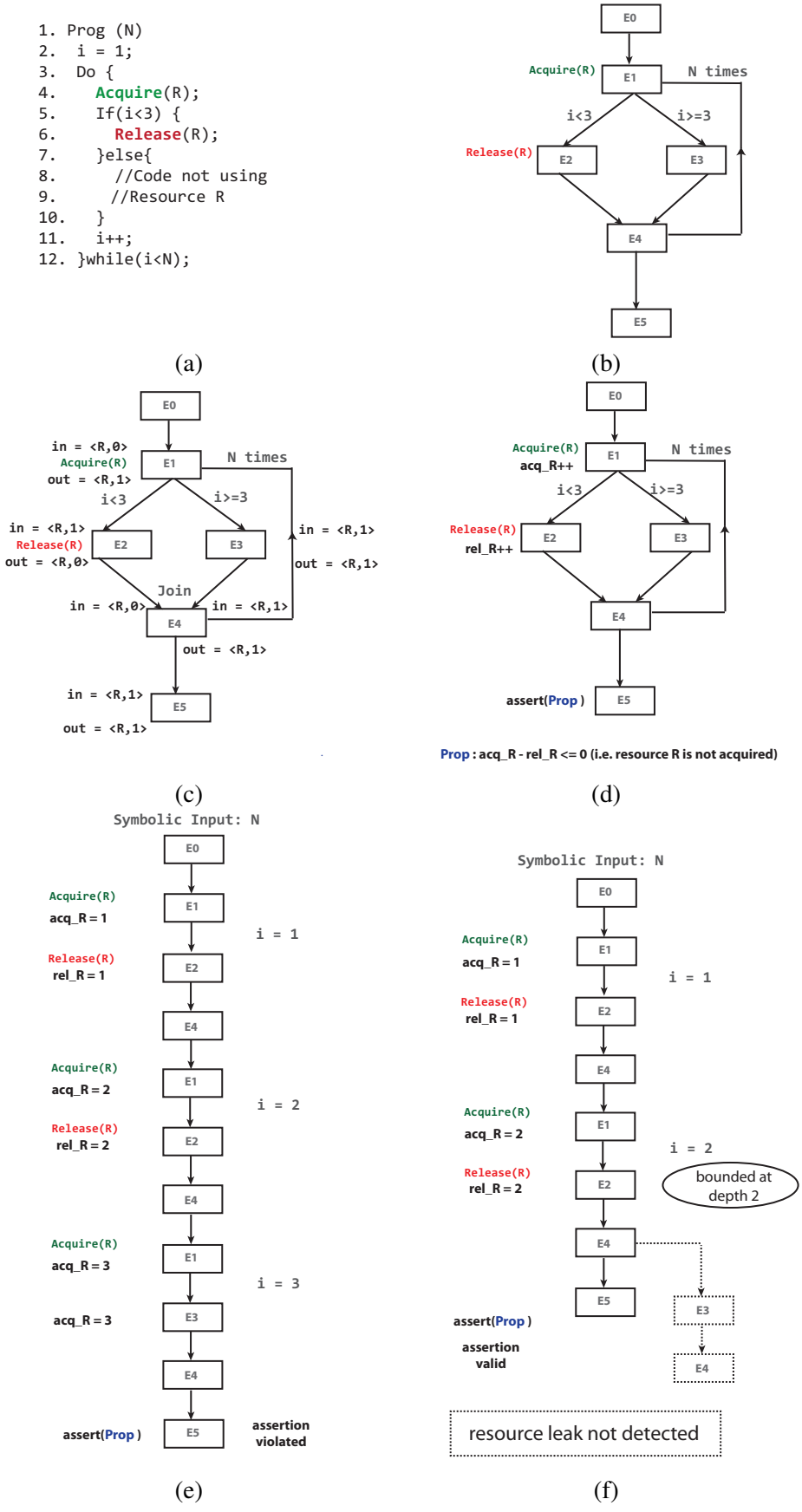


Figure 38: Overview by example (a) example code with a potential resource leak (b) CFG othe example code (c) static analysis of the code. Input and output abstract states are shown for each node in the graph (d) assertion added to the exit node of the graph (e) symbolic exploration and test case generation (f) limitation while using bounded symbolic execution

execution (on the property of interest) along a program path. More specifically, the *update* operation reflects the effect of executing an instruction over the abstract state and has the type of Equation 34.

$$U : D \times P \rightarrow D \tag{34}$$

The abstract join operation (J), on the other hand, is used to combine two abstract states into one. In our implementation, the join operation computes the least upper bound on the abstract domain (D). Whenever there are multiple abstract states coming from different control flows into a program point, we use the abstract join operation to combine them. The abstract join operation has the type of Equation 35.

$$J : D \times D \rightarrow D \tag{35}$$

The property of interest in our framework is shown in Property 1. A resource can be present in either of these two states : *acquired* or *not acquired*. Furthermore, if a resource is reference-counted, it is associated with an integer that is incremented whenever the resource is acquired, and decremented whenever the resource is released. If there are one or more resources that are in the *acquired* state at the end of the detection phase we report the potential for an energy bug.

Property 1. *All energy-intensive, hardware resources and power-management utilities should be in the* released *state at all exit nodes of the (analysed) app.*

Resources in the Android applications are represented by Java objects. Therefore, we shall first define the semantics for Java object tracking in subsection 6.4.1. Subsequently, we shall extend this representation for resource tracking in subsection 6.4.2.

6.4.1 Java Object Tracking

To reliably track Java objects, we need to have a domain D, that abstracts the various memory structures containing objects. We will need to represent (1) the set of object references (O), (2) the Java stack (s) and (3) the variables (v).

1. Let $O = J \cup \{\top\}$ be the set of abstract object references. The set J represents the set of concrete Java object references. Element $o \in O$ either represents a concrete Java object, or is equal to $\{\top\}$ (any Java object).

2. Let $s : \mathbb{N} \rightarrow O$ be a function representing an abstract stack state, and let S be the set of abstract stack states. For any stack state s, the value $s(0)$ represents the top-most (most recently pushed) element, the value $s(1)$ represents the element pushed before $s(0)$ and so on.

3. Let $v : E \rightarrow O$ be a function representing the abstract variable states, and let V be the set of such states. The set E represents the set of possible Java variable expressions. An element in E can refer to a local variable, a member field, or a static field.

The abstract domain (D) for tracking Java objects is of type as shown in Equation 36, where $\mathcal{P}(O)$ represents the power set of all abstract object references, S represents the abstract stack state and V represents the abstract variable states.

$$D = \mathcal{P}(O) \times S \times V \tag{36}$$

6.4.2 Resource Tracking

In order to track a resource object we need to extend the domain of D with the state of the resource and the set of possible acquire locations. The resultant abstract domain D' is shown in Equation 37, where K represents all possible states for a resource and P represents the set of all program points. A resource can either be in *acquired* or *not acquired* state. If the resource is reference counted, its state is equal to the upper bound on the acquire count, or to the value $+\infty$, if it cannot be statically bounded. Therefore, the resource state K equals the set $\mathbb{N} \cup +\infty$.

$$D' = D \times K \times P \tag{37}$$

Now since our abstract domain has been defined we can further elaborate on the nature of the abstract operations for resource tracking. The update operation (U') at a program point P, can be represented by Equations 38 and 39. Here $d' \in D'$ and $d' = \langle d, k, p \rangle$, where $d \in D$, $k \in K$ represents the state of the resource and p is the set of acquire locations.

$$U' : D' \times P \rightarrow D' \tag{38}$$

$$U'(d', P) = \begin{cases} U \bullet U_{res}(d', P) & \text{, if instruction at } P \text{ is an acquire} \\ & \text{or release instruction} \\ (U(d, P), k, P) & \text{, otherwise} \end{cases} \tag{39}$$

U denotes the abstract update operation for an instruction that is not related to resource acquire or release. The symbol $\bullet$ denotes function composition. The operation U_{res} in Equation 39 is invoked whenever a resource acquire or release instruction is encountered. The function of U_{res} is as follows. Whenever we encounter an instruction for acquiring a resource r, we add P to the set of acquire locations for r. Additionally, if the resource r is reference counted we increase $k_r (\in k)$ by one, otherwise we set k_r to 1. On encountering a release instruction, we reduce $k_r (\in k)$ by one, if $k_r > 0$.

The join operation (J) can be represented by Equation 40 and 41, where we join resources from two sets ($D_1, D_2 \in D'$). If both sets contain the same resource (*i.e.* associated with the same Java object), we take the maximum of the reference counts (Max operation in Equation 41), and we merge the acquire-location sets. For all other cases (*i.e.* when $d_1 \neq d_2$) we abstract the Java object to *top* ($\top$, represents the largest element) and add to the resultant set.

$$J : D' \times D' \rightarrow D' \tag{40}$$

Let $D_1 = \langle d_1, k_1, p_1 \rangle$ and $D_2 = \langle d_2, k_2, p_2 \rangle$.

$$J(D_1, D_2) = \begin{cases} \langle d, Max(k_1, k_2), p_1 \cup p_2 \rangle & \text{if } d1 = d2; \\ \langle \top, k_1, p_1 \rangle \cup \langle \top, k_2, p_2 \rangle & \text{otherwise.} \end{cases} \tag{41}$$

6.4.3 Detecting Potential Energy Bugs, Instrumenting Assertions

As a result of the abstract interpretation analysis, we can get the abstract state at each program point. Let $\langle d, k, p \rangle \in D'$ be the abstract state at the end of the program. Then k represents the state of resources at the end of the program. An abstract state at the end of the program with $\exists (k_r \in k), k_r > 0$ denotes that resource r that may have been acquired but not released on some path, in the program. In other words, Property 1. is violated. Such a scenario implies the presence of a potential energy bug.

To detect resource leaks, our framework automatically instrument Property 1. as assertions at all exit node of the (analysed) app. The exact instrumentation slightly differs for reference counted and non-reference counted resources. However, in the both cases we first instrument two new counter variables acq_r and rel_r for each (potentially) unreleased resource r. The instrumentation is such that the variable acq_r is increased every time resource r is acquired and variable rel_r is increased every time resource r is released. For a reference counted resource, the assertions is such that it checks the value of formula $acq\_r - rel\_r \leq 0$ *i.e.* there are at least as many releases as acquires for the resource r. Whereas for a non-reference counted resource the assertions checks the value of formula $acq\_r \neq 0 \wedge rel\_r = 0$ *i.e.* there are one or more acquires but no releases for resource r. Once instrumented these assertion are tested for violations in the validation phase.

6.5 VALIDATION

The potential energy bugs detected in the previous phase are validated in this phase. In this phase, we use a symbolic execution based technique to test the assertions instrumented in the previous phase. It is worthwhile to know that symbolically executing the entire application may often be impractical due to the issue of *state-space explosion*. Therefore, before symbolically exploring a potentially buggy application, we apply a couple of search space reduction techniques to reduce the number of program paths that need to be explored (in ordered to validate or invalidate the instrumented assertions). The search-space reduction techniques, namely (a) Transitive closure computation of EFG and (b) Symbolic input reduction, are discussed in Section 6.5.1. Subsequently, test input generation process is detailed in Section 6.5.2. The complete flow of the validation phase is shown in Figure 39.

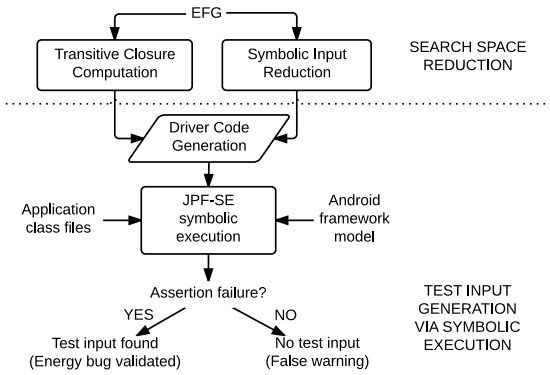


Figure 39: Overview of the validation process

6.5.1 Search Space Reduction

Transitive closure computation of EFG: The event flow graph (EFG) of an application captures all events in an application. However, some of the events (represented by nodes in the EFG) may not influence the acquiring of a resource in any feasible execution. Therefore, such events (EFG nodes) can be excluded during exploration. The nodes that need to be explored are grouped into two sets. The first set (S_1) consists of all nodes that fall on a path from an entry node to the resource acquiring node. The second set (S_2) consists of all nodes that fall on a path from the resource acquiring node to an exit node. All nodes that are not contained in these two sets (*i.e.* $S_1 \bigcup S_2$) need not be explored symbolically. This computation

is repeated for all resources that may lead to potential energy bugs (as reported in detection phase). Figure 40 shows an example of transitive closure computation. In Figure 40(a) there are three paths from entry node ($E1$) to exit node ($E6$), however only the path $E1 - E3 - E6$ is of interest for checking the validity of Property 1.

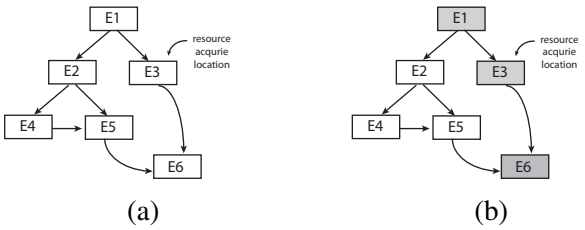


Figure 40: Example of transitive closure computation. EFG node $E3$ is resource acquire location. Transitive closure computation gives the list of nodes shown in shaded in (b)

Symbolic input reduction: Techniques based on symbolic exploration often face the issue of scalability whenever there are a large number of symbolic program states to be explored. The number of symbolic states to be explored is directly influenced by the number of symbolic inputs to the program. Since Android applications are *event-driven*, inputs to an application may arrive during execution. As shown in Figure 36, the two potential inputs for an Android application are *(i)* the return value of an Android API call, and *(ii)* the arguments supplied to an application-level event handler when the Android system invokes the callback routine (due to an event trigger). A typical Android application may receive many such inputs (*e.g.* the application may frequently invoke Android APIs and read the return values). As the values of the inputs are not known statically, we have to treat the input values as symbolic during exploration. However, exploring the program with all input variables made symbolic may be very expensive (or even impractical). To alleviate this issue, we selectively make an input variable symbolic only if that input-variable may affect the execution of program paths where the potential resource leaks are reported.

To realize this, we use an existing *static program slicing* technique to capture the set of instructions in a program that may influence the execution of the program towards an energy bug, due to a resource R. Specifically, for each acquire(release) site of resource R, we statically compute a *backward slice* with respect to the instruction that acquires(releases) R. Subsequently, only the input variables that are used by any instruction captured in any of the computed slice have to be made symbolic. Figure 41 shows a simple example of slicing algorithm shown in Algorithm 2. The bold lines (figure on the right) shows the slice after application of Algorithm 2. The inputs to this example program are `i1`, `i2`, `i3` and `i4` while `v1` and `v2` are local variables. For the resource to be acquired and not released, only input variables `i1` and `i2` (source code lines numbers $1, 2, 3, 8, 9$ and 10) are relevant. It is worthwhile to note that the entire else branch (lines 13 - 19) is irrelevant as no resource is ever acquired if the execution comes to the else branch.

6.5.2 Test Input Generation

The final step in the validation phase is the generation of test inputs that expose assertion violation (and hence energy bugs) in the analysed app. For this purpose we use the tool JPF-SE [139], which is a symbolic execution extension for the tool Java PathFinder (JPF). It is worthwhile to know that Android apps, unlike conventional Java programs do not have a *main* method as the starting point of execution. Instead the execution of an Android application

Algorithm 2 Slicing for relevant inputs in a program

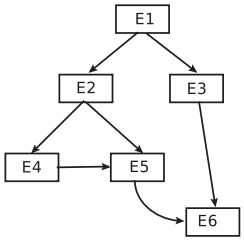
1: **Input:**
2: R: resource that may be involved in an *energy bug*
3: **Output:**
4: V: the set of input variables to be made symbolic
5:
6: $T \leftarrow \{\}$
7: **for all** acquire sites of resource R, I_{acq} **do**
8: $S \leftarrow$ compute backward slice w.r.t. instruction I_{acq}
9: $T \leftarrow T \cup S$
10: **end for**
11: **for all** release sites of resource R, I_{rel} **do**
12: $S \leftarrow$ compute backward slice w.r.t. instruction I_{rel}
13: $T \leftarrow T \cup S$
14: **end for**
15: $V \leftarrow$ PARSESLICE(T)
16:
17: **function** PARSESLICE(*slice*)
18: **return** set of all input variables that are used in any instruction contained in *slice*
19: **end function**

```
1 if (i1 == 0) {
2    if (i2 == 0) {
3         v1 = 0;
4    }
5    if (i3 == 0) {
6         v2 = 0 ;
7    }
8    acquire(R);
9    if (v1 == 0) {
10        release(R);
11   }
12 }
13 else {
14   if(i4 == 0) {
15        release(R) ;
16   } else {
17        v1 = 0 ;
18   }
19 }
```

Set of relevant inputs: i1, i2

Figure 41: An example showing how our slicing algorithm (Algorithm 2) works.

starts from a *root* UI screen. JPF-SE however works for conventional JAVA programs only, therefore our framework automatically generates a driver file that represents the structure of the analysed app's EFG. The generation of the driver code is a straightforward process. The first event handler to be called in the driver is that of the *root* UI screen, followed by its child nodes (event handlers). In the scenario where an EFG node E contains multiple child nodes $c_1, c_2, \ldots, c_i$, we create conditional branch statements for each child node c_i. The execution of a conditional branch statement is decided based on a newly added variable $ctrl\_E$. Essentially, the variable $ctrl\_E$ represents the event (or user input) that decides the execution of a child node at E. While executing the application symbolically, we make the variable $ctrl\_E$ symbolic. This allows us to explore all possible event sequences at a given EFG node in the application. Figure 42 shows an example for driver code generation.



(a)

```
void E1() {
    /* Invoke E1's event handler here */
    if (ctrl_E1 == 0) {
        E2();
    } else {
        E3();
    }
}
void E2() {
    /* Invoke E2's event handler here */
    if (ctrl_E2 == 0) {
        E4();
    } else {
        E5();
    }
}
...
```

(b)

Figure 42: An example for driver code generation

As described in Section 6.4.3, for all potential energy bugs reported by the detection phase, we instrument assertions at the exit points in the EFG. Symbolically executing through the application via the driver code allows us to check the validity of the instrumented assertions. Each assertion violation is recorded and the corresponding failure revealing test-cases is presented to the developer as a witness for the reported bug.

Following sequence provides an example of a bug-revealing test sequence generated by our framework for the app Tachometer. The bug-revealing test sequence contains all the information that the developer needs to replicated the reported bug. For instance in the following example the bug-revealing scenarios tell about the UI events and their relative ordering that needs to be triggered to observed the reported bug. In addition, the framework also reports the event-handler signatures (shows in square brackets in the following example), to further assist the developer. It is worthwhile to know that a single user event can trigger multiple event-handlers, such as in the following example, event `id/button1/TAPSCREEN_120_93` triggers event-handler `WahlActivity$1_onClick` followed by `PositionActivity_onCreate`.

```
Buggy Sequence:
  entryNode/KEYPRESS_82 [WahlActivity_onCreate]
  -> id/button1/TAPSCREEN_120_93 [WahlActivity$1_onClick]
  -> [PositionActivity_onCreate]
  -> MenuButton/KEYPRESS_82/
  -> BackButton/KEYPRESS_4/
  -> [PositionActivity_onPause]
```

Figure 43: Test case generated for app Tachometer

6.6 AUTOMATED REPAIR

In the final phase, our framework automatically generates repair expressions for the validated energy bugs. Figure 45 shows the work-flow of this final phase. To generate a repair expression, we need to determine (i) the repair expression and (ii) the repair location. The repair expression is affected by the choice of repair location and hence we first discuss how the repair location is obtained.

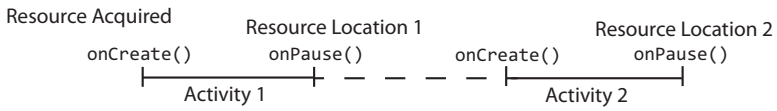


Figure 44: An example scenario

It is worthwhile to mention that the objective of the repair phase *is primarily to fix the reported energy bug*. In some scenarios there might be multiple locations at which the repair expression can be added. Techniques such as those discussed in [59] can be used for finding the *optimal* repaired program from a set of repaired programs. However, for our work we would only consider repairing such that the repaired program does not fail at the energy bug revealing test case. For example, in the simplistic scenario shown in Figure 44, the repair code can be either put at $RepairLocation1$ or at $RepairLocation2$, depending on whether or not $Activity\ 2$ needs the resource acquired in $Activity\ 1$. For energy efficiency, an acquired resource should be released as soon as it is not required anymore. However, determining whether an acquired resource is still needed may not feasible just by analyzing the instructions of the application code. This is because for certain resources, such as Wakelocks, *last-use* information is not explicitly found in the application code (a Wakelock prevents the CPU from going to sleep as long as it is acquired). In such cases, we choose a conservative repair strategy for our framework. The repair expression is always put in the last method ($onPause$) of the exiting activity in the bug revealing test case (generated by the validation phase). Our strategy is always guaranteed to fix the energy bugs as witnessed by the test case, but the automatically generated repair may not be optimal under all circumstances.

<resource_expr> . <release_API_call> (42)

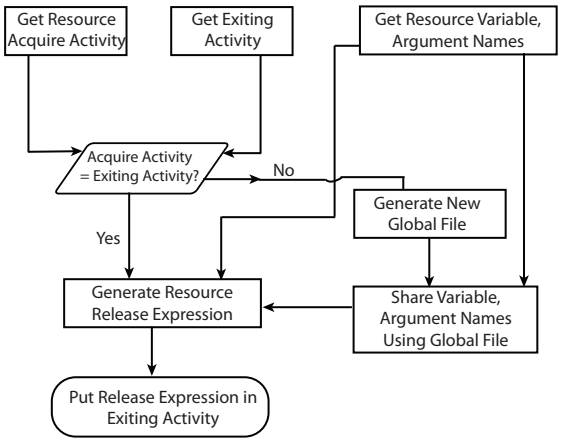


Figure 45: Work flow for automated repair in our framework

The repair expression automatically generated by our framework has the format shown in Equation 42. resource_expr in Equation 42 denotes the expression to access the resource object at the repair location and release_API_call represents the API call to release the resource object. To form a syntactically correct repair expression we need to obtain the variable name for the resource object and the arguments to the release API call. These information are obtained from the result of the detection phase (described in Section 6.4).
In the scenario where resource acquiring activity and exiting activity are different (such as in the example of Figure 44), our framework adds additional pieces of code to ensure syntactic correctness. In particular, a new global file is automatically added by our framework. This global file is used to share the resource variable name and parameters to release API call from the resource acquiring activity to the exiting activity. This work-flow is also shown in Figure 45.

6.7 ECLIPSE PLUGIN ENERGYPATCH

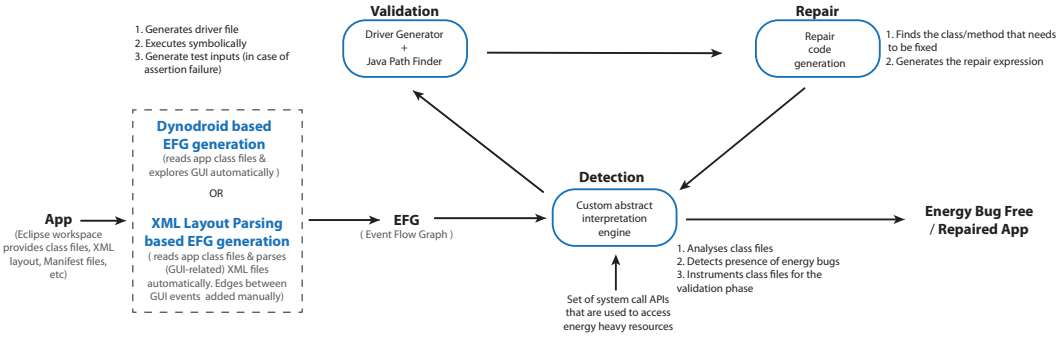


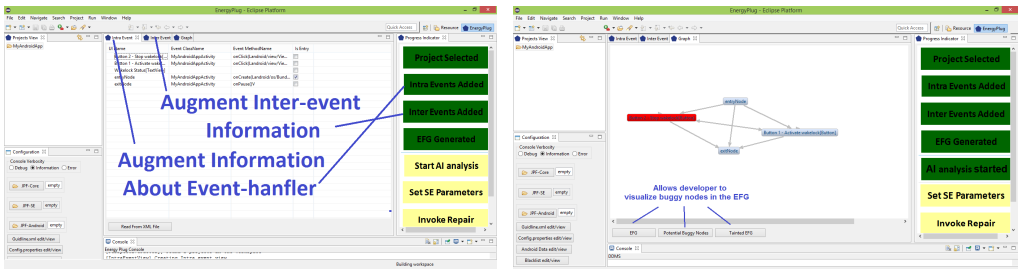
Figure 46: Work flow inside EnergyPatch

Our framework has been implemented into an Eclipse plugin named EnergyPatch. This source code for the plugin is available under the BSD 3-Clause license [140] from BitBucket[137]. In this section we shall briefly describe the structure and working of the tool. An interested reader can find the detailed instructions for installation and usage at `http://www.comp.nus.edu.sg/~rpembed/epatch/home.html`.
Figure 46 shows the work flow of EnergyPatch. In the plugin, the EFG generation can be done by using an automatic GUI exploration tool Dynodroid [81] or by parsing the GUI-related XML files from the app (In Android, it is common practice to specify the GUI layout of the app by means of XML files). We have also added an option (cf. Figure 47 (a)) that allows the developer to manually augment any additional flow dependencies (intra or inter event) within the EFG. This can be an useful feature in the case where the automatic GUI exploration misses any event in the EFG during exploration.

For the detection phase, we have implemented an abstract interpretor for Java bytecode (for app class files). The abstract interpretation based analysis also requires the set of resources and associated API calls that need to be tracked with in the app. This information is also provided through an XML file and can be modified/replaced by the user as required. In case the abstract interpretor does not find any energy bugs, the analysis stops. However, in the case where potential energy bugs are found, the framework alerts the developer of same. This potential energy bug information is mapped to the EFG of analysed app and displayed in the

graph view of the tool (Figure 47(b) shows an example). Such a pictorial representation may further assist the developer in understanding the debugging process.

In case a potential energy bug is found, the validation phase generates the driver code as described in Section 6.5.2. Also the required class files (from the analysed app) are instrumented using ASM [141], as described in Section 6.4.3. Subsequently, the app (after search space reduction as described in Section 6.5.1) is executed symbolically. For Symbolic execution, our tool relies on Java Pathfinder (JPF), more specifically three components of JPF: JPF-core [142] , JPF-SE [143] and JPF-Android [144]. Where JPF-core provides the base JPF classes, JPF-SE provides the symbolic execution support and JPF-Android provides the model of Android framework. During symbolic execution, all assertion violations are reported to the developer. All reports are accompanied by a witness test-case, that can be used to replicate the said energy bug, using a mobile device and a power meter. Subsequently, the repair expressions are generated and presented to the developer as described in section 6.6.



(a) (b)

Figure 47: Screenshot of EnergyPatch (a) shows how developer can manually augment EFG (b) visualization inside tool showing information such as the structure of the EFG, buggy nodes, etc

6.8 EXPERIMENTAL EVALUATION

Experiments for the evaluation of framework answer the following research questions: *(i)* Efficacy of our framework *i.e.* how effective is our framework in uncovering test cases that lead to real *energy bugs (ii)* Importance of the detection phased in our framework *(iii)* Effectiveness of the automated repair i.e. does the repair expression generated by our framework actually makes the application more energy efficient? and finally *(iv)* Comparison of our framework with existing works on resource leak detection in mobile apps. We shall discuss these research questions in Sections 6.8.2 - 6.8.5. However first we shall discuss the experimental setup and choice of subject apps in Section 6.8.1.

6.8.1 Experimental Setup

We created a suite of subject apps using 35 apps available from online sources such as Google Play, F-droid app repository, Github and Google Code. These apps are diverse in terms of functionality, complexity, application size and popularity (based on user ratings and number of downloads for apps available from Google Play). It is worthwhile to know that energy bugs can manifest only in those apps where energy-intensive Android API calls have been used (as discussed in Section 6.2.3), therefore, while creating the suite of subject apps we only

Table 10: Subject apps for which energy bugs have been reported through bug-reports and/or previous publications

App Name, version / code	App Description	LoC / Size(KB)	Event Handler Classes	Defect Description
Aripuca, 1.3.4 / 24 [108]	Recording tracks and saves waypoints	8093 / 660	14	Moving from MainActivity to WaypointActivity causes location updates to stay on even after the app is paused
□ Omnidroid, 0.2.1 / 6 [123]	Automated event/action manager for Android	12425 / 258	28	Location updates started by the app are not stopped after app is paused or phone is restarted
Tachometer 1.0 / 1 [121]	App to measure location and speed	793 / 540	9	Selecting PositionActivity from main screen of the app causes location updates to be acquired but not released
△, □ Bablesink 1.0 / 1 [145]	An app to help locate lost phones	521 / 21	1	An exception may cause the app to have a potential wakelock bug
Sensor Tester 1.0 / 1 [126]	Sensor monitoring and logging app	1719 / 400	6	App acquires location and sensor services without releasing them on app pause
Aagtl 1.0.31 / 31	Geocaching based app for Android	20572 / 307	4	Pressing Home button during cache download causes the wakelock to remain acquired by the app
□ DroidAR 1.0 / 1 [146]	An augmented-reality app for Android	18177 / 398	6	Going to the ArActivity then switching back to another activity causes the GPS to stay on even after closing app
Benchmark 1.1.5 / 9 [147]	Benchmarking app for Android	9739 / 1020	23	Navigating from Benchmark activity to show results causes the wakelock to be not released by the app
♢, □ Osmdroid 3.0.1 / 2[148]	Provides replacement for Android's MapView	8107 / 276	10	Selecting the sample loader followed by first sample causes the app to not release the location updates
□ Recyclelocator 1.0 / 1 [149]	Area-specific restroom, mailbox finding app	717 / 116	3	Location services are not disabled when the map module is paused as a result GPS is constantly looking for a signal
□ SP Transport 1.17 / 18 [150]	Android app that assists in bus-travel	1766 / 161	3	Defective behaviour observed in the LocationView class, GPS is never turned off when the activity is paused
△, □ Ushaidi v2.2 / 13 [151]	App for Collection, visualization for crisis data	10621 / 713	22	CheckinMap keeps the GPS on, even after the user has navigated away from the activity
□ Zmanim 3.3.84.296 / 84 [152]	List of halachic / halakhic times	72977 / 842	4	GPS signal acquisition from the ZmanimActivity is never stopped even after the app is paused

□: app used in [83] △ : app used in [76] ♢ : app used in [77]

choose apps that have usage of atleast one energy-intensive component (such as wakelock, GPS, Wifi, etc). In addition, we also try to make the suite of subject apps diverse by including apps that use different kinds of energy-intensive resources (such as wakelock, sensor, GPS, Wifi, etc). Additionally, the test-suite also includes apps that were used in our previous works such as [2] and other related works such as [83], [76] and [77]. This will allows us to compare our framework with existing related works. Table 18 lists down a few details of the subject apps which contains energy bugs. These details include the app description, app size (LoC), number of event handlers and defect description for each of these apps. In Table 18, apps used in [83] are marked using the symbol □, apps used in [76] are marked using the symbol △ and apps from [77] are marked using the symbol ♢.

Our test-generation and repair framework was implemented in Java. It was run on a on a Desktop-PC with an Intel Core i7-2600 CPU (quad-core) with 8GB of RAM and Ubuntu 14.04

Table 11: Results of Detection/Validation phase for app listed in Table 18

App name	Resources Not Released	Detection Phase		Validation Phase	
		Time (s)	Scenario	Time (s)	Event-handlers Invoked
Aripuca	GPS	21	Activity Switching	53	17
Omnidroid	GPS	3	Activity Switching	44	3
Tachometer	Sensor GPS	<1	Resource Acquire Loop	12	6
Babblesink	Wakelock	<1	Uncaught Exception	2	n/a (app crashes)
Sensor Tester	Sensor	4	Resource Acquire Loop	32	7
	GPS		Activity Switching		
Aagtl	Wakelock	4	Activity Switching	28	4
DroidAR	GPS	6	Activity Switching	51	5
Benchmark	Wakelock	4	Activity Switching	3	3
Osmdroid	GPS	5	Activity Switching	14	6
Recycle-locator	GPS	1	Activity Switching	3	4
SP Transport	GPS	2	Activity Switching	3	5
Ushaidi	GPS	4	Activity Switching	4	6
Zmanim	GPS	7	Activity Switching	34	5

OS. The mobile device used to run app was an off-the-shelf LG Optimus E400 smartphone. This mobile device was running an Android Gingerbread(v2.3.6) operating system(OS), which was the most widely used OS at the time of these experiments. It is worthwhile to know that newer versions of Android such as Android Jelly Bean and Ice Cream Sandwich have similar API calls (for resource usage) to that of Android Gingerbread (v2.3.6), therefore our framework should work equally well for app intended for these platforms as well. Finally, for measuring energy savings in the patched apps we used a Yokogawa WT210 digital power meter using the a setup shown in Figure 57.

6.8.2 Efficacy of Our Framework

The most important research question in the evaluation is about finding out the efficacy of our framework in finding and reporting energy bugs in real-life Android apps. To answer this we ran our framework for all subject apps (including the apps listed in Table 18) to observe, (i) if our framework could detect energy bugs and (ii) whether the test-cases generated our framework can be used to replicate these reported bugs on a real mobile-device.

In our experiments, our framework reported bugs (with test-cases) for 12 of the apps. Among these 12 apps, 8 had energy bugs involving GPS (not all apps use the same APIs for accessing the GPS), 2 apps had energy bugs involving both the GPS and Sensors and 2 apps had energy-bugs due to improper usage of Wakelocks. *It is important to know that our framework reports the presence of energy bugs in an app only after both the detection (static analysis) and validation (symbolic execution) phases have been completed.* It is also worthwhile to know that our framework has a relatively less performance overhead as both the computationally intensives phases *i.e.* the detection and validation phases, were completed in approximately a minute even though some of the application were significantly large with thousands of lines of code. This goes on to show that our framework can be applied to energy bug detection in real-life apps. When we manually inspected the apps for which potential energy bugs were detected we observed following three scenarios:

i. *Activity Switching*: a resource is acquired in an activity, however the app navigates to another activity or stops execution without releasing the acquired resource.

ii. *Resource Acquire Loop*: a resource is repeatedly acquired within a loop however it is not released a sufficient number of times before exiting the application.

iii. *Uncaught Exception*: unexpected execution flow in the program due to uncaught exception may leave resource(s) in the acquired state.

To check the usefulness of the generated test-cases we manually replayed these test-inputs on the test device and compared the resource states using the debugging tool Android debug bridge. By doing this additional step we were able to confirm that the test cases do indeed lead to buggy scenarios. Additionally, for some apps such as *DroidAR*, *Osmdroid*, *Recycle-locator*, *SP Transport* and *Ushaidi* there exists user reports (on code repositories) describing the user-observed energy-related defect. For these apps, we were able to compare the test-cases generated by our framework to the test-scenarios reported by the user. We observed significant similarities in these comparisons as well. In the following subsection, we shall describe our observations from the analysis for two apps *Sensor Tester* and *Babblesink* in more details. For one these apps, *Sensor Tester* our framework generates failure-revealing test-case whereas for the other *Babblesink* our framework detected potential for an energy bugs but still did not generate any failure-revealing test-cases.

Case Studies

Sensor Tester[153] app allows its user to monitor and log data from Sensor/GPS on the mobile device. An user can start/stop logging data from the sensor by simply toggling the "Logging" button. For this app, the detection phase reported two potential bugs, one due to Sensor and another due to the GPS. Further evaluation with JPF-SE validated the presence of these bugs. JPF output (test cases) suggest that while the application is logging data, pressing the "Back" button exits the application but does not release the acquired resources. Additionally, every time the user re-enters the application and restarts logging, a new sensor connection is established, while the previous sensor connections keep getting accumulated. We also observed that the GPS stays acquired when the application exits. Testing this app with the bug-revealing test inputs on a fully charged smartphone caused the battery to completely drain in less than eight hours (standby time of the test device is approximately 600 hours[136]).
Babblesink [145] app allows the owner of a lost phone to locate it. For this app, the detection phase of our framework reported a potential Wakelock related energy bug. However, further analysis during the validation phase failed to produce a feasible test case that triggers the reported bug. In the app, a Wakelock object is acquired to ensure the interruption-free initialization of an *IntentService*. Ideally, the Wakelock should have been released after the initialization of the *IntentService* is completed. However, there exists a path in the application that will bypass the release instruction. This path is executed when an exception occurs after the acquire instruction for the Wakelock object has been executed but before the execution of the release instruction. This bug was initially reported by [76]. We manually inspected this app to find out the reason due to which the validation phase did generate any valid test-case. We observed that the exception that caused the release code to be skipped actually crashes the app because it is uncaught up to the top level function of the application. Since all acquired Wakelocks are released in the event of an app crash, no actual energy bug can occur in any feasible execution scenario therefore the validation phase couldn't generate any test-case for this app.

6.8.3 Importance of Detection Phase in the Framework

We have emphasized in previous sections (*c.f.* Section 6.3), that the use of static analysis in the detection phase make our framework scalable and also helps in reducing the overall analysis time. Here we present some observations to support these claims. We compare the (Symbolic

Execution) SE only approach to the (Abstract Interpretation + Symbolic Execution)AI+SE approach for uncovering energy bugs, where

- *SE only* approach implies only symbolic execution is used to uncover energy bug(s) without the preceding static analysis
- *AI+SE* approach (our approach) implies that we perform static analysis (Abstract Interpretation or AI) followed by validation (Symbolic Execution or SE)

Specifically, we conducted experiments for two scenarios: *(i)* analysis time for both approaches in the absence of energy bugs, and *(ii)* analysis time for both approaches if energy bugs do exist. For the scenario where no energy bugs exist, static analysis terminates relatively fast (less than 15 seconds) when using the *AI+SE* approach (the one implemented in our framework). Additionally, since the results of the detection phase are always *sound*, we can be assured that no energy bug indeed exists at least for the portion of app represented by its EFG. However, to come to the same conclusion using *SE only* approach, all feasible program paths must be explored. Since there can be an unbounded number of event sequences in an app (because UI elements in the app can be repeatedly navigated), the SE only approach can potential take forever to conclude.

For the second scenario (where at least one energy bug exists in the analysed app), the *AI+SE* approach can produce results in up to one-third of the time of *SE only* approach for certain apps (*e.g.* validation time for *Omnidroid [154]* was 117 seconds for SE only as compared to 44 seconds for the AI+SE approach). This difference in evaluation time happens because the detection phase of our framework helps in search space reduction. The magnitude of search space reduction is directly influenced by the program location at which the (energy-bug-causing) resource/utility is acquired. The farther the (energy-bug-causing) program location is from the root UI node, the more the gains by using our search space reduction technique.

6.8.4 Effectiveness of Automated Repair

Our framework uses the test-cases generated by the validation phase to generate the repair expressions (described in Section 6.6). For instance, the test case for the app *Tachometer* that is shown in Figure 43, is used to generate the repair for class `PositionActivity.java` as shown in Figure 48.

```
@Override
public void onPause() {
    Log.i("PositionActivity", "onPause");
    /* repair expression start */
    this.locationManager.removeUpdates(this);
    /* repair expression finish */
    super.onPause();
}
```

Figure 48: Repair expression for app Tachometer

To evaluate the effectiveness of the repair, we compared the energy consumption of the original app to that of the repaired app, for the buggy test-input. The setup for energy-measurement used a test device (LG Optimus E400 smartphone running Android v2.3.6) and a power meter as shown in Figure 29. The power meter used in our experiments was Yokogawa WT210 that has approximate sampling rate of 50 KS/s . Energy consumption of the device is measured for a period of 300 seconds *after* the bug revealing test-case has been executed. This is done

to measure the impact of the buggy app code on the energy-consumption behaviour of the device. It is worthwhile to know that the test-cases were executed manually. Also no other apps were being executed on the test-device while the power-measurement experiments were being conducted. Power measurements were conducted thrice for each experiment and the average value for reading were computed. Additionally, during these experiments the *screen timeout* duration of the device was set to 30 seconds. Table 12 shows the increase in energy efficiency of the buggy apps before and after the repair has been applied.

Table 12: Improvement in energy consumption of all apps with validated energy bugs after the automatic repair

App Name	Energy Consumption (J)		Avg. Improvement %
	Before Repair	After Repair	
Aripuca	161.3	89.0	44.8
Omnidroid	103.4	89.3	13.6
Tachometer	224.3	89.5	60.1
Sensor Tester	205.9	88.5	56.0
Aagtl	91.7	74.6	18.6
Benchmark	125.1	81.6	34.7
DroidAR	192.4	76.1	60.4
Osmdroid	197.5	75.7	61.6
Recycle-locator	186.8	76.9	58.9
SP Transport	208.2	84.2	59.5
Ushaidi	217.4	85.0	60.9
Zmanim	197.8	79.5	59.8

6.8.5 Comparison with Existing Works

The works of [76], [77] and [83], are most related to our current work as they all presents techniques for detecting and/or characterizing resource leaks in mobile apps. Therefore, in this subsection we shall discuss how our technique compares to the technique presented in these works. Additionally, we shall see how effective our framework is when analysing the subject program used by these works.

The works of [76], [77] use static analysis to detect resource leaks in Android apps. In particular, [76] proposes a technique based on data-flow analysis, whereas, the technique presented in [77] is based on function call graph traversal. Both of these works have observed that their technique may produce some false positives. Unlike our framework, the techniques used in these works do not have a dynamic analysis or a test-generation phase, as a result there is no mechanism to automatically prune out the false positives that may be introduced due to the over-approximations in the static analysis phase. One such false positive was observed for the app *Babblesink* which was described in the section 6.8.2. Unfortunately, we could only obtain the source-code for three apps from [76] and [77] (some apps of [77] were closed source whereas some programs used in [76] were individual class file from older versions of the Android framework). Those apps for which we could obtain the source-code, our framework was able to successfully find bugs in two of them (*Osmdroid* and *Ushaidi*) while for the third app *Babblesink* our framework reported that no feasible test-cases was present to trigger a potential resource leak. More details of the *Babblesink* analysis can be found in Section 6.8.2.

The work in [83] proposes a dynamic analysis based technique for resource leak detection. In particular, it uses bounded symbolic execution for finding test-cases that lead to resource leaks. In general, bounded symbolic execution implies that the depth at which symbolic exploration takes place is bounded. Bounding the depth of symbolic exploration may create limitations of its own. For instance, if in the example of Figure 38(f), the (loop) bound for exploration is set to 2 iterations, symbolic exploration would be unable to find any resource

leaks. In general, knowing the adequate bound (such that all bugs can be revealed) can be quite challenging. Therefore, using only symbolic execution may not be optimal strategy for exploration. On the contrary, in our framework we first use static analysis to *conservatively*, detect the presence (or absence) of resource leak, after which we use symbolic execution to generate test-cases. From the work of [83] we were able to obtain eight apps, seven out of which we were able to analyse successfully with our framework. For the eighth app, *Babblesink*, our framework did not produce any test-cases for reasons mentioned in Section 6.8.2.

6.9 THREATS TO VALIDITY

One of the major threats to the results produced by our framework is due to the incompleteness of the EFGs used in our analysis. In our framework, EFG of an app is generated using a dynamic analysis technique. Since we cannot guarantee the completeness of the EFG, therefore, we cannot provide any completeness guarantee for the generated results as well. As a consequence of this limitation, in case of incomplete EFG our framework may leave portions of the app code unanalysed that have not been represented in the EFG.
Another source of threat to the results generated by our framework arises due to the use of Android framework model. In our framework we use an Android framework model in order to ensure the correct execution of the app. We have based our Android framework model on an existing Android model proposed in [144]. It is worthwhile to mention that defects in the Android framework model can affect the results of our framework, so we invest additional effort to ensure that the framework model works appropriately.

6.10 CHAPTER SUMMARY

In this chapter, we presented a framework that can provide an end-to-end solution for detecting, validating and repairing energy bugs in real-life mobile apps. The use of light-weight static analysis technique in the detection phase allows us to quickly narrow down the potential program paths along which energy bugs may occur. Subsequent exploration of these potentially-buggy program paths using dynamic analysis technique helps us in validating these potential energy bugs. Our framework also generates test-cases for all validated energy-bugs which can be used by the app-developer to manually recreate the buggy scenarios. Finally, our framework generates repair expression to fix the validated energy bugs. We implemented our framework as an Eclipse plugin so that it can be easily installed and used by app-developers during app-development. We conducted experiments to evaluate the effectiveness and efficacy of our framework by testing real-life Android apps. During these experiments our framework reported energy bugs in twelve out of thirty-five tested apps. Also the test-cases generated by our framework were able to trigger the reported energy bugs on a real mobile-device. Finally, we compared the energy-consumption of the buggy apps post repair on our test-device. In these experiments we observed that the repair code generated by our framework can improve the energy-efficiency of the buggy apps significantly.

7 AUTOMATED RE-FACTORING OF ANDROID APPS TO ENHANCE ENERGY-EFFICIENCY

Mobile devices, such as smartphones and tablets, are energy constrained by nature. Therefore, apps targeted for such platforms must be energy-efficient. However, due to the use of energy-oblivious design practices, often this is not the case. In this chapter, we present a light-weight, re-factoring technique that can assist in energy-aware app development. Our technique relies on a set of energy-efficiency guidelines, that encode the optimal usage of energy-intensive (hardware) resources. Given a prototype for an app, our technique begins by generating a *design-expression* for it. A *design-expression* can be described as a regular-expression representing the ordering of energy-intensive resource usages and invocation of key functionalities (event-handlers) within the app. It also generates a set of *defect-expressions*, that are *design-expression* representing the negation of energy-efficiency guidelines. A non-empty intersection between an app's *design-expression* and a *defect-expression* indicates violation of a guideline (and therefore, potential for re-factoring). To demonstrate the efficacy of our technique we analysed a suite of open-source, Android apps with our technique. The resultant re-factoring when applied, reduced the energy-consumption of these apps between 3 % to 29 %. We also present a case study for one of our subject apps, that captures its design evolution over a period of two-years and more than 200 commits. Our framework found re-factoring opportunities in a number of these commits, that could have been implemented earlier on in the development stages, had the developer used an energy-aware re-factoring technique such as the one presented in this work.

7.1 INTRODUCTION

Easy access to app-development tools (such as Eclipse ADT[155]) and a low barrier to entry[1] has led to an abundance of mobile apps in recent days. As of year 2015, there were more than 1.8 million apps available on Google Play Store [157] alone. A plethora of online tutorials and publicly available testing tools, such as MonekyRunner[158], make it relatively easy, even for new app-developers to develop and test the functionality of their apps. However, the same cannot be said for the non-functional behaviour of apps, specifically energy-efficiency. Mobile devices, such as smartphones and tablets, are energy-constrained by nature. Therefore, it is important that apps targeted at such platforms be designed and optimized for energy-efficiency. However, due to a combination of factors such as, lack of proper understanding of energy-efficient designs and a lack of tools that can enforce such energy-efficient designs, app development has mostly been done in an energy-oblivious manner.

In recent years, research works have proposed a number of techniques (such as profiling [3], testing [2]) that can be used post development, for quality assurance purposes. Such techniques however, do not provide adequate support for energy-efficient design and development of apps. In this chapter, we present an orthogonal (and complimentary) approach, to address this issue. We present a light-weight, re-factoring technique that uses a set of energy-efficiency guidelines, to generate energy-efficient re-factorings for a given app. These energy-efficiency guidelines were formulated under the assumption that energy-efficiency can be increased by optimizing the usage of energy-intensive (hardware) resources. Resources such as I/O Components and power management utilities have the biggest impact on energy-consumption,

1 Registration fees for a publisher account at Google's Play store costs 25 USD [156]

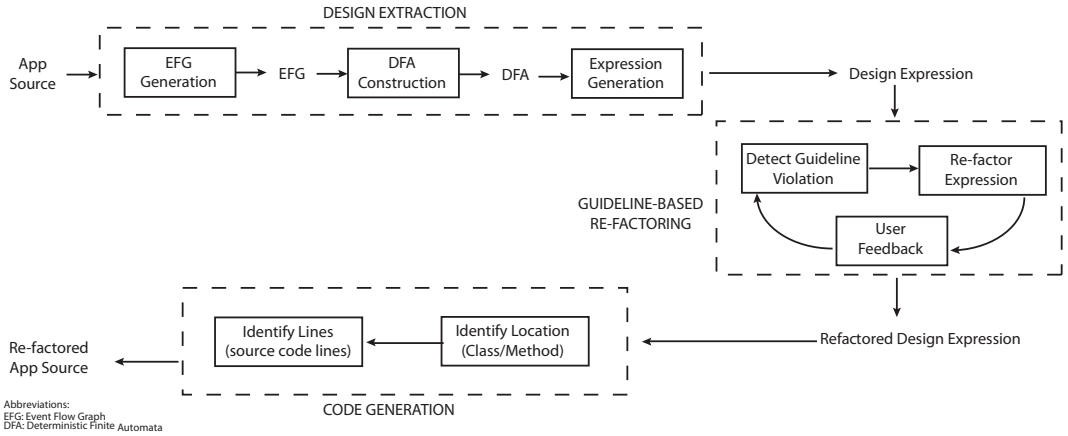


Figure 49: An overview of the re-factoring framework

hence their usage must be reduced as much as possible, without affecting the functionality of the program. Additionally, certain resources (such as sensors) can be accessed through multiple configurations, each of which provide specific trade-offs between Quality-of-Service and energy-efficiency. Judicious usage of less-expensive resources, based on the functionality of the app can further decrease energy consumption.

To detect re-factoring opportunities, our framework checks for violations of (energy-efficiency) guidelines in a given app. However, doing so directly on the app source-code may be in-appropriate for a number of reasons. For instance, mobile apps being event-driven in nature, usually consists of segregated pieces of code (or event-handlers), ordering between which may not be explicit in the app-source code. This makes it difficult to detect guideline violations across event-handlers boundaries. Additionally, real-life apps may contain thousands of lines of code, not all of which affect the energy-consumption behaviour of the app significantly. Therefore, before our framework looks for re-factoring opportunities, it first generates an intermediate, *succinct* representation of the app. This intermediate representation, henceforth referred to as *design-expression*, contains only those information which is most relevant to the energy consumption behaviour of the app. More formally, a *design-expression* can be described as a regular expression, that represents the ordering of energy-heavy, resource usages and invocation of key functionalities (event-handlers), within the app. The use of *design-expression* allows us to re-factor energy-intensive resources across event-handler boundaries. Additionally, since *design-expression* are customized regular expression we can use off-the-shelf tools and techniques to analyse/manipulate them. It is also worthwhile to know that our framework generates the *design-expression* for a given app *automatically*.

In order to detect guideline violations our framework also generates a set of *defect-expressions*. A *defect-expression* has same syntax as that of *design-expression* but represent the negation of an energy-efficiency guideline. So essentially, *design-expression* represents what an app is supposed to do (in order to achieve its functionality) whereas the *defect-expression* represents what an app is *not* supposed to do in order to be energy-efficient. A non-empty intersection between *design-expression* and *defect-expression* indicates violation of the energy-efficiency guideline that is associated with the *defect-expression*. It is worthwhile to know that such an analysis is possible because both the *design-expression*, as well as the *defect-expression* are constructed from the same alphabet. On detecting a guideline violation, our framework generates a re-factored *design-expression* such that it has an empty intersection with the *defect-expression*. Finally, the re-factored *design-expression* is presented to the

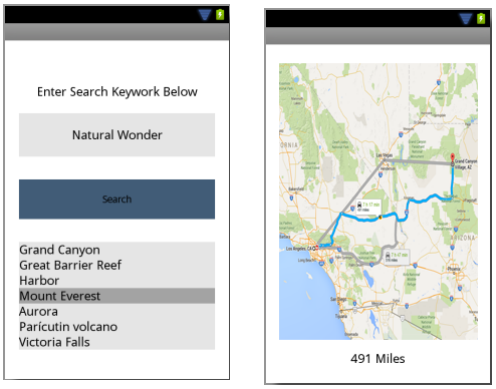
app-developer for approval. If the developer approves the presented re-factoring, the changes are mapped back to the source code.

7.2 OVERVIEW

Our framework is composed of three key components (overview shown in Figure 49): (i) design extraction component (ii) re-factoring component and (iii) code generation component. The objective of design extraction component is to generate the design-expression for the app. The most crucial processing happens in the re-factoring component, where the design-expression is evaluated for guideline violation and design expression re-factoring takes place (if any guideline violations are detected). Finally, the code generation component maps the changes from the re-factored design-expression to the app source code. These components are discussed in detail in sections 7.2.2 - 7.2.4, with the help of an example-app that is described in Section 7.2.1.

7.2.1 Example App

To keep the proceeding discussion concrete, we shall explain the overview of our framework using an example-app. Let us consider a simple app that allows its user to search for famous landmarks based on provided keywords. If the user selects any of the landmarks, the app shows the landmark on a map, along with the distance of the device to the selected landmark. The app has local copies of all the information (landmark names, coordinates, map tiles, etc) that is required to do its computation, except for the user/device location. The user/device location is obtained through an on-board GPS receiver. The app initiates the location updates as soon as it is started and the location updates are stopped only when the app exits (the foreground). The screen-shots provided in Figure 50 can provide a rough idea about the graphical user interface (GUI) layout of the app. It is worthwhile to know that location-updates are one of the most energy-intensive operation on a mobile device and hence it should be used for as small duration of time as possible. However, in this example-app location-updates have been used sub-optimally. More specifically, the location updates are active for the entire duration of time the app is active (in the foreground), whereas the location-updates are used only when the user selects a landmark (*i.e* when Screen 2 is shown). Through our framework we wish to detect and re-factor instances of energy-inefficient behaviour, such as sub-optimal resource binding as present in this example-app.



Screen 1 Screen 2

Figure 50: An example app

7.2.2 Design Extraction

Our framework begins analysis by generating an appropriate intermediate representation of the app that is to be evaluated. Performing re-factorings directly on the app source code would be in-appropriate for a number of reasons. For instance, mobile apps being event-driven in nature,

usually consists of segregated pieces of code (or event-handlers), ordering between which may not be explicit in the app-source code. This makes it difficult to detect energy-inefficient patterns across event-handlers. Additionally, real-life apps may contain thousands of lines of code, not all of which affect the energy-consumption behaviour of the app significantly. Therefore, we create a succinct intermediate representation of an app which contains only the information that is most relevant to its energy consumption behaviour.

Energy consumption in mobile apps has a direct co-relation to the use of Android API calls that are related to acquire, usage and release of energy-intensive resources [3]. Previous works such as [159] have found that the Screen, Wifi, GPS, Sensors, Camera, CPU and Keypad are some of the most energy-intensive resources on a mobile device. Hence, acquire, usage and release API calls for these energy-intensive resources are included in the intermediate representation. Additionally, the intermediate representation also captures the different user-interaction patterns by which an user can interact with the app. Our objective, after all, is to re-factor an app so as to remove (or at least minimize) the user interaction (UI) patterns that may lead to energy-inefficient behaviour. Considering all these requirements we create the notion of *design-expression*.

Definition 7.2.1 *A design-expression is a regular expression which represents the ordering of Android API calls (acquires, release & usages) for energy-intensive hardware resources and invocation of event-handlers within an app.*

A design-expression is similar to a regular expression in terms of syntax and expressibility. Like a regular expression, a design-expression is constructed with symbols and operators. The symbols of the expression are user-inputs (such touches, taps, etc) while operators are the same as regular grammar (*eg.* $*$ implies 0 or more). A detailed discussion on regular expression grammar can be found in [160]. The key advantages of using design-expression can be summed up as follows:

- It is a succinct representation for an app and contains only that information which is most relevant to its energy consumption behaviour. It is worthwhile to know that design-expression can be used to represent the set of all input strings that can be used to interact with the app.
- Since design-expressions are based on regular expressions we can use a wide-variety of existing tools and techniques that are applicable to regular expressions, to manipulate design expression (such as minimizing an expression or computing the intersection of two expressions, etc).

Generation of the design-expression from app sources takes place in three steps: (i) EFG Generation, (ii) DFA Construction and (iii) Expression Generation.

***(i) EFG Generation*:** An event-flow graph (EFG) [88] can be used to represent the GUI model of an app and can be defined as in Definition 7.2.2. Figure 52 shows a simplified EFG for the example-app of section 7.2.1. The GUI states A and D correspond to app start and exit states. While the states B and C correspond to the app being in Screen 1 and Screen 2, respectively (*cf.* Figure 50). The events a and d represent the starting and closing of the app. Whereas the event b represents the user pressing search button and the event c represents the user selecting a landmark. Since an user can repeatedly press the search button on the screen 1 therefore there is a self-loop at EFG node B. It is worthwhile to know that EFGs for real-life apps can be more complicated because of the omnipresent UIs such as the Back button and the Menu button. However, for the purpose of simplicity we shall not include these UIs (Back and Menu button) in the EFG of Figure 52. Finally, the Android API calls x_r, u_r

and y_r represent the acquire, usage and release of resource r, (in the example of section 7.2.1 it is location updates).

Definition 7.2.2 *An event-flow graph is a directed graph, that captures all possible event-sequences that can be used to interact with an app. The nodes of an EFG represent GUI states. A directed edge between two nodes of an EFG X and Y represents that state Y follows state X. Additionally, nodes of the EFG are annotated with event-handler information associated with their respective events.*

In order to generate the EFG we use an automated, GUI exploration tool Dynodroid [81]. Dynodroid uses a publicly-available, Android tool Hierarchy Viewer [101], to obtain the UI layout of an app. It then uses this layout information to progressively explore all the UI states of an app. By extending Dynodroid we can obtain the events as well the directed edges between the GUI states. We also need to obtain the event to event-handler mapping information for EFG generation. This information can either be obtained by modifying the Android platform or instrumenting the apk files. We choose the later because it is more straightforward and maintainable as it need not be re-implemented every time the Android platform is updated. In particular, the instrumentation is done for event-handlers that are defined in the `android.app.activity`, `and roid.app.service` and `android.content.Broadcast Receiver` packages of the Android framework. We also obtain an event-handler to Android API call mapping for all the energy-intensive resources by statically analysing the bytecode of an app. For instance, invocation of API call `com.google.android.maps.MyLocationOverlay.enableMyLocation` would be recorded as an acquire for the resource GPS in the event-handler where the API call was used. It is also worthwhile to mention that this event-handler to Android API call mapping is done in an object-insensitive manner. For instance, in this location example all invocation of the API call `com.google.android.maps.MyLocationOverlay.enableMyLocation` would be allocated to the same GPS resource. Finally, the event-handler to Android API call mapping is combined with the event to event-handler mapping as shown in Figure 51, to generate the EFG.

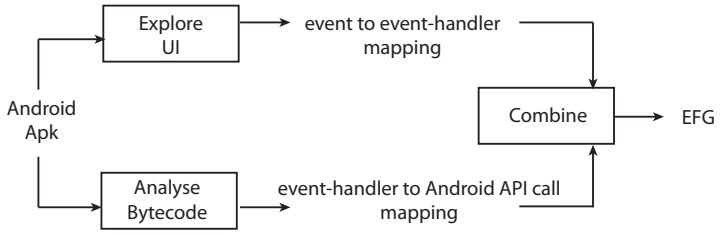


Figure 51: Event-flow graph (EFG) generation

***(ii) DFA Construction*:** Once the EFG is obtained, it is converted into a deterministic finite automata (DFA). This conversion is done so that we can use standard algorithms to do DFA-to-Expression generation. The DFA is constructed such that each node in the DFA either represents the starting of (execution of) an event-handler, stopping of (execution of) an event-handler, acquiring of a resource, release of a resource or usage of a resource. In the scenario where an EFG node is not associated with a resource-related Android API call, the conversion from EFG node to DFA node is straightforward. However, in the scenario where an EFG node does contain resource-related Android API calls, the EFG node is divided into multiple DFA nodes (depending on the number of Android API calls contained in the EFG

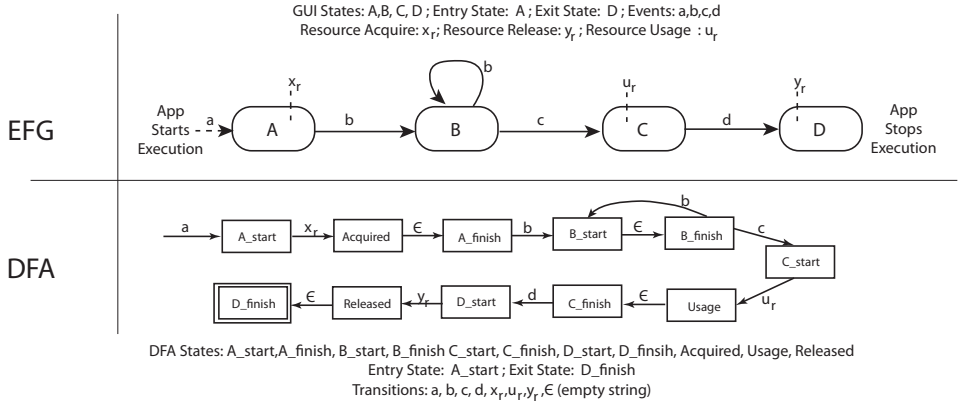


Figure 52: Event-flow graph (EFG) and deterministic finite automata (DFA) for the example-app of section 7.2.1

node). Finally, the entry states and the exit states are copied from the EFG to the DFA. Figure 52 shows the DFA for the example-app of section 7.2.1.

***(iii) Expression Generation*:** Post DFA construction, we extract the design expression (*i.e.* the regular expression) representing the DFA. The conversion from DFA to expression is done using the standard algorithm as proposed in [160]. Essentially, the algorithm proceeds by removing the DFA states (and changing the transitions accordingly), until only initial and final states are remaining. The resultant expression is subsequently minimized using an off-the-shelf Python library [161]. For instance, the resultant design-expression for the example-app of section 7.2.1 is $ax_rb^*cu_rdy_r$.

7.2.3 Guideline-based Re-factoring

The re-factoring component of our framework operates in two steps: detecting guideline violating patterns and re-factoring. To detect guideline-violating patterns in an app's design expression, our framework first generates the defect-expression (for each guideline). A defect-expression can be described as a design-expression representing the negation of a guideline. A non-empty intersection between the (app's) design-expression and defect-expression indicates the presence of a guideline violating pattern. It is worthwhile to know such an analysis is possible because the design expression and defect expression are constructed using the same alphabet.

Consider the example-app from section 7.2.1 where early-resource binding (for location updates) takes place. Assume that the guideline ϕ represents the fact that resource binding should happen as late as possible, then $\neg\phi$ (defect expression) represents its negation *i.e.* the scenario where early resource binding takes place. The information that there is potentially long delay (on node B) between the acquire and usage of resource r, can be easily obtained through our framework. In particular, for the example-app of section 7.2.1 design expression and defect expression ($\neg\phi$) are shown in expressions 1 and 2, respectively. Here $\bullet$ implies all feasible symbols and $\neg$ implies negation (of an symbol). Operators $*$ and $+$ represents 0 or more times and 1 or more times, respectively. A non-empty intersection between expression 43 and expression 44 (shown in expression 45), provides an evidence for guideline violation.

$$\text{Design Expression} \quad : \quad ax_r b^* cu_r dy_r \tag{43}$$

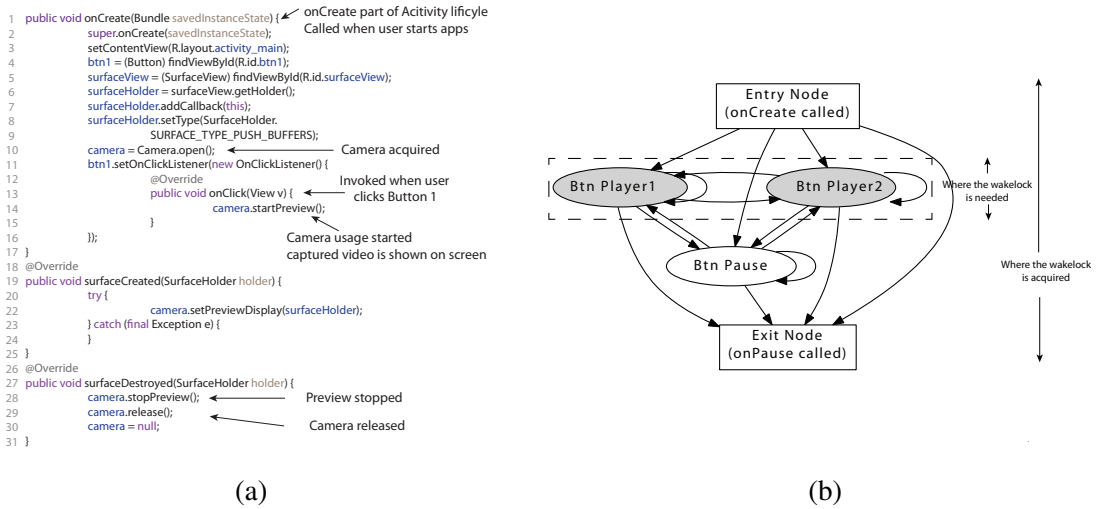
$$\text{Defect Expression} \quad : \quad \bullet^* x_r[\neg u_r][\neg u_r]^+ u_r[\neg y_r]^+ y_r \bullet^* \tag{44}$$

$$\text{Intersection} \quad : \quad ax_r b^+ cu_r dy_r \tag{45}$$

$$\text{Re-factored Expression} \quad : \quad ab^* cx_r u_r dy_r \tag{46}$$

In a scenario such as in this example, where guideline-violation is detected, the app's design expression is re-factored such that the resultant design-expression has an empty intersection with the defect expression. The re-factoring method depends on the specific guideline that has been violated (further described in section 7.3.2), however, following observations can be stated for all re-factored design-expressions.

- *Guideline Conforming*: The intersection between the re-factored design expression and the defect expression is empty. (For example, re-factored design expression shown in expression 46 has an empty intersection with the defect expression shown in expression 44)
- *Functionality Preserving*: The re-factoring is such that the original functionality is preserved. In particular, re-factoring affects the position of resource acquire and/or release (symbols) in the design expression. However, the resource usage (symbols) are left untouched. Additionally, the relative ordering between resource acquire, usage and release ($acquire \Rightarrow usage \Rightarrow release$) is always ensured.



(a) (b)

Figure 53: (a) A code fragment showing sub-optimal camera binding, (b) Sub-optimal Wakelock acquisition in app ChessClock

7.2.4 Code Generation

Once the design expression has been re-factored and the changes approved by the app-developer, we can map the changes back to the source-code. This is done in two steps: identifying the re-factored location (source-code line numbers within event-handlers) and implementing the changes to new location. As described in the previous section, re-factoring initially happens at the level of design expression, where each symbol in the design-expression corresponds to some event or acquire/usage/release of a resource in the app. It is worthwhile to

know that re-factoring only affects the position of resource acquire/release (relative to events) in the design expression. Therefore, by comparing the original design-expression to the re-factored one we can identify the event-handlers that need to be modified. More specifically, two sets of event-handlers are modified: event-handlers where the resource acquire/release Android API call used to be (in the original expression) and event-handlers where the resource acquire/release Android API call need to be (obtained from the re-factored expression). For instance, observe that the resource acquire symbol, x_r, in expression 46 is moved after event c suggesting that Android API call for resource acquire should be moved to the event-handler for invocation of GUI state C (original location of Android API call represented by x_r was in GUI state A). It is worthwhile to know that we record the event-handler to source-code mapping information during the EFG generation step. Therefore, we can easily identify the event-handlers that need to be modified to implement the re-factoring. Additional (syntax-level) information that may be needed to conduct the re-factoring (such as parameters to the Android API call) are obtained through flow-analysis on the original source code.

Table 13: Configuring resources for different QoS and energy-efficiency

	Location Updates (GPS/Network)	Sensor Updates (accelerometer,orientation, etc)	Wakelocks (Screen, CPU, Keypad)
High Power Consumption	gps_provider	sensor_delay_fastest	full_wake_lock, screen_bright_wake_lock
Moderate Power Consumption	network_provider	sensor_delay_game	screen_dim_wake_lock
Low Power Consumption	passive_provider	sensor_delay_normal, sensor_delay_ui	partial_wake_lock

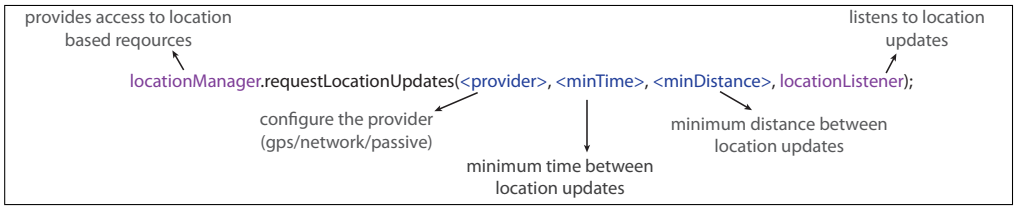


Figure 54: Various parameter that affect QoS, energy-consumption for location updates

7.3 GUIDELINE-BASED RE-FACTORING

The basic premise on which these guidelines for energy-efficiency have been formulated is the fact that minimizing usage of energy-intensive resources increases energy-efficiency of an app. The resources in question being energy-intensive hardware components such as GPS, Camera, Wifi, Bluetooth, Sensors, *etc* or power management utilities such as Android Wakelocks. We shall first discuss the guidelines in section 7.3.1 and the algorithm for guideline-based re-factoring in section 7.3.2.

7.3.1 Energy-efficiency Guidelines

The guidelines can be stated as follow:

1. *Sub-optimal Bindings*: Resources must be acquired as late as possible (during the execution of the app) and released as early as possible.

2. *Nested Usages*: Nesting of resources (acquire-releases) should be avoided.

3. *Trade-offs, QoS Vs Energy-efficiency*: Certain information (such as location updates), can be obtained through multiple resources, each varying in quality-of-service (QoS) and energy consumption. If the application functionality permits, QoS can be traded-off to improve energy-efficiency.

4. *Resource Leaks*: All resources acquired during the execution of app must be released before the app exits.

***Sub-optimal Bindings*:** Roughly translated, this guideline implies that resource acquire, usage and release should be as close to each other as possible. However, due to the event-driven nature of mobile-apps, source-code proximity may not necessarily imply closeness. Consider the code fragment shown in Figure 53(a). This app has the basic functionality to capture images through camera and display them on the screen when user clicks an on-screen button. In this example, the camera is acquired (line 10) when the user starts the app (*i.e.* in function $onCreate$). However, there might be a substantial delay between the resource being acquired and resource being used ($camera.startPreview()$ on line 14). This is because the preview is only started when the user clicks the button, thereby triggering the event handler defined on line 11. The period of time in between the app start and user event, is the time when the camera is consuming energy needlessly and can be avoided. It is worthwhile to know that for certain resources, such as Wakelocks, resource usages cannot be explicitly associated with any API calls (*i.e.* such resources only have acquire and release API calls). For such cases, developer help may be needed to identify the functionalities (event-handlers) that utilize the resource. For instance, as shown in the Figure 53(b), the app ChessClock [162] requires the Wakelock to be acquired when either of the two players are interacting with the app. However, in the app, Wakelock is acquired for entire duration of app activity.

$$\text{Defect Expression} \quad : \quad \bullet^* x_r [\neg u_r][\neg u_r]^+ u_r [\neg y_r]^+ y_r \bullet^* \tag{47}$$

To generate the defect expression (needed for detecting guideline-violation in design expression), we identify (and use) the symbols associated with the resource acquire, usage and release in the design expression. Expression 47 shows an example defect expression representing sub-optimal binding for the example-app of Section 7.2.1. where $acquire(r)$, $usage(r)$ and $release(r)$ for resource r are denoted by symbols x_r, u_r and y_r, respectively. In the scenario, where the intersection between defect expression and design expression is non-empty, the design expression is re-factored. The re-factoring is such that the (symbols for) resource acquire/release are re-arranged to be as close to resource usage in the design expression. However, during the re-factoring relative ordering, between the acquire, usage and release is always maintained.

***Nested Usages*:** As stated by guideline *Sub-optimal Binding*, resources must be in the acquired state for the smallest period of time possible to achieve the app functionality. Complementary to this guideline is *Nested Usage* guideline, which states that nesting of resources should be avoided. In particular, this guideline applies to those resources which generate same (type of) information. The utility of this guideline is that, if at any stage during re-factoring nesting of resource usages is observed in the design expression, the expression can be simplified by removing the nesting so as to reduce the duration of time for which the acquired resource is active. For example, in the code fragment shown in Figure 55, from app *Sensorium* [163] (commit hash:$9d141b7$), the API call $requestLocationUpdates$ is invoked twice. However, since both the invocations of API call provide location updates, these two invocations can be merged into one, without loss of functionality.

```
1 public void enable() {
2   locationListener = new LocationListener() {
3                 //....
4  };
5  locationManager = (LocationManager) context.getSystemService(Context.LOCATION_SERVICE);
6  locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER, 0, 0,  locationListener);
7  locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0, locationListener);
8  enabled = true;
9 }
```

Figure 55: Code-fragment from Sensorium showing nested resource usage

In expression 48 we show an example of defect expression generation for detecting the presence of nested usage scenarios. Expression 48 is constructed in context of example-app from Section 7.2.1, where $acquire(r)$ and $release(r)$ for resource r are denoted by symbols x_r and y_r, respectively.

$$\text{Defect Expression} \quad : \quad \bullet^* x_r [\neg y_r]^* x_r \bullet^* \tag{48}$$

***Trade-offs, QoS Vs Energy-efficiency*:** Mobile app functionality is often based on sensor information (such as acceleration, orientation, location, *etc*) collected from the physical environment. To obtain this information an app must interact with the available I/O components. Since mobile OSs, such as Android, were designed to run on energy-constrained device, they provide a number of ways (API call configurations; *cf* Figure 54) to interact with the I/O devices. For instance, in Android most of the power-hungry hardware components (GPS, sensor, screen, etc), can be operated at various levels of power consumption and Quality-of-Service (QoS). In this context, higher QoS implies more precise data, at higher update-frequency. In general, higher QoS leads to higher power consumption.

Table 13 shows a list of a few such configurations that can be used in combination with API calls to obtain desired QoS. For instance, column 1 of Table 13 lists three different variations of location updates that can be used along with API call $requestLocati\,onUpdates$. Both $gps\_provider$ and $network\_provider$ actively initiates a location fix, when invoked. Whereas, $passive\_provider$ doesn't initiate a location fix actively, but provides a location fix by passively listening to location updates from other apps (on the device). As a result, it has a significantly less impact on the energy-consumption of the device. However, since the information generated by $passive\ \_provider$ can be stale, it is only suitable for apps that require a rough estimation of user's location (for e.g. news apps). Similarly, when comparing between $gps\_provider$ and $network\_provider$, the former provides more precise location update (suitable for travel apps), whereas later is less-precise but relatively more energy-efficient (suitable for continuous location based content generation). It would be impractical to use one set of configuration for all apps, however, once the app-developer provides the app-category (such as used in Play store [164]), appropriate energy-efficient re-factoring can be suggested. In our framework, we look for such configuration in the app source code, with the help of Apache Lucene [165] search libraries.

***Resource Leaks*:** This is one of most commonly occurring and (energy) expensive defects in mobile apps. Essentially, any resource acquired during the execution of the app must be released before that app ceases to execute. However, since real-life apps have many potential exit locations, ensuring resource releases at all such exit locations may be challenging (specially for apps with many GUI screens). In the presence of such a defect, the device (more specifically the unreleased resource) keeps consuming energy even after the defective app ceases to execute.

It is possible to view the scenario of *resource leaks* as a very extreme case of *sub-optimal resource binding*, where the acquired resources are released after infinite period of time. However, we categorize them in two different categories so as to keep the analysis straightforward. For an app to have a resource leak, there should be at least one path within the app, triggered by a sequence of user events, that ends with an unreleased resource. For instance, defect expression representing resource leaks for the example-app of Section 7.2.1 is shown in expression 49, where $acquire(r)$ and $release(r)$ for resource r are denoted by symbols x_r and y_r, respectively.

$$\text{Defect Expression} \quad : \quad \bullet^* x_r [\neg y_r]^* \tag{49}$$

7.3.2 Guideline Implementation

An important question that may arise at this point is how to enforce the energy-efficiency guidelines as described in Section 7.3. The two approaches that we can think of for implementing these guidelines would be to either embed them in the platform itself (by means of OS manipulation, middlewares, *etc*) or to enforce them through energy-aware re-factoring tools that assist the app-developer during the development process. It is worthwhile to know that the *middleware approach may be unsuitable for real-life apps as it may make the platform inflexible*. This is because the functionalities of real-life apps may vary widely and are usually subjected to developer discretion. For instance, faster, energy-hungry sensor updates are unsuitable for battery life (such as stated in QoS Vs energy-efficiency trade-offs guideline), however, the app-developer may still want to use it. Similarly, all acquired resources should be released before the (resource acquiring) app leaves the foreground (as stated in resource leak guideline), but the app-developer may choose not to do so (say for instance the app wants to log the user whereabouts throughout the day using location-updates). In contrast, our *re-factoring framework approach is much more flexible as it allows the developer to choose which of the suggested re-factoring are to be applied* to the source code. Additionally, since all the changes are made only to the app and not to the platform, *the finished app should behave similarly across all devices* (OS/middleware is usually customized by the device-vendor).
We use Algorithm 3 to implement the energy-efficiency guidelines described in Section 7.3.1. Algorithm 3 takes in an app (source-code as well as GUI-layout files) and its category as inputs and generates the re-factored design expression, Ref_{app}. It begins by generating the design-expression (procedure `GenerateDesign`) as described in section 7.2.2. This design-expression is then checked for guideline violations using procedure `CheckViolation`. The procedure `CheckViolation` returns a tuple $\langle v, r \rangle$, where $v \in V \bigcup \{NoDefect\}$ and $V = \{ SubOptimalBinding, TradeOff, NestedUsage, ResourceLeak. \}$, whereas, r is the resource that participates in guideline violation v. If a guideline violation is detected (*i.e.* $v \in V$), we proceed to re-factor the design-expression based on the type of guideline violation. To being re-factoring, we first extract the symbols associated with acquire (X_r), usage(U_r) and release(Y_r) of resource r from the design-expression using the procedure `GetResSymbols`. The procedure for re-factoring depends on the type of guideline violation. In case of suboptimal resource binding guideline violation, the symbols in X_r are inserted (in the re-factored expression) before symbols in U_r. This operation is done using the procedure `insertBefore`. Similarly, procedure `insertAfter` is used to insert symbols in Y_r after symbols in U_r. In case of nested usage guideline violation, the symbols in X_r are merged into a single symbol using the procedure `merge`. Similar merging is done for symbols in Y_r. In case of QoS/efficiency trade-off guideline violation, the design expression stays the same, however, the API calls associated with symbols in X_r are re-configured based on provided app category ($AppCt$). Finally, in case of resource leak guideline violation, the symbol for releasing

Table 14: Key results. For each app, we provide app-description, size metrics, observed defects and energy-saving observed as result of applying the re-factoring suggested by our framework

Name(Version)	App Description	Apk Size (KB)	LoC	Energy Saving	Re-factoring Description
Sensorium (1.1.12)[166]	Collect sensor data	1248	4001	21	Restricting use of sensors/GPS to key functionality. Adding resource release at exit.
UserHash (1.1[167])	Location reporting service	171	837	15	Adding GPS release at exit.
Aripuca (1.3.4)[108]	Tracking app	660	8093	15	Adding GPS release at exit.
ShareMyPosition (1.0.11)[168]	Share your location	25	474	3	Replacing Full Wakelock with less-expensive counterparts.
DroidSat (2.47)[169]	Satellite Viewer	146	15007	4	Removing nesting of location resources. Replacing GPS uses with less-expensive counterparts.
iTLogger (1.0.0)[170]	Speed/heading information	553	4014	9	Replacing Full Wakelocks with less-expensive counterpart. Adding GPS, Sensor release at exit.
Heart Rate (1.0)[171]	Heart rate monitor	849	557	5	Replacing Screen Bright Wakelocks with less-expensive counterparts.
ChessClock (1.2.0)[162]	Touchable chess clock timer	336	725	14	Restricting use of Wakelock to key functionality. Replacing Full Wakelocks with less-expensive counterpart.
0xBenchmark (1.1.5)[172]	Mobile benchmark suite	1020	9739	29	Restricting use of Wakelock to key functionality. Adding resource release at exit.
Ham (1.5.7) [173]	Amateur radio tools	43	2224	6	Replacing GPS uses with less-expensive counterparts.

resource r is added after symbols in U_r. The final re-factored expression (Ref_{app}) can be used to map the changes back to the source-code. It is worthwhile to know that Algorithm 3 can re-factor resource acquires/releases across event-handler (class/method) boundaries. This increase the re-factoring opportunities drastically, however, this may also cause some syntax-level inconsistencies (such as due to modifiers associated with variables). However, since our re-factoring technique is made for design/development stages of app-development such inconsistencies can be removed with developer assistance.

```
1   @Override
2   public void onResume() {
3     //...
4     long minTime = 6000;
5     float minDistance = 10;
6     locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,
7                                            minTime, minDistance, locationListener);
8     //...
9
10  }
```

These lines are moved together, if the requestLocationUpdates needs to be re-factored to another location

Figure 56: Re-factoring while maintaining flow-dependencies

It is worthwhile to know that in the original source code, resource related lines may have flow dependencies. To discover and preserve these dependencies we perform flow analysis using the tool Soot [174]. For instance, in the example shown in Figure 56, the invocation of API call *requestLocationUpdates* (line 6) depends on the value of the long variable *minTime* (line 4) and float variable *minDistance* (line 5). In such a scenario, if the re-factoring requires moving the invocation of *requestLocationUpdates* to another location, all related lines (*i.e.* lines $4-7$) are also moved. To identify the re-factoring location (class/method where re-factored lines will be moved/added), we check the position of the resource-related symbols with respect to user-event related symbols,in the re-factored design expression.

Algorithm 3 Re-factoring design expression

Input:
App: App source files
$AppCt$: Category such as Games, Travel, Books, *etc*
Output:
Ref_{app}: Re-factored design expression

$Des_{app} \leftarrow$ `GenerateDesign(`App`)`
$\langle v, r \rangle \leftarrow$ `CheckViolation(`$Des_{app}, App, AppCt$`)`
while $v \in V$ **do**
    $\langle X_r, U_r, Y_r \rangle \leftarrow$ `GetResSymbols(`Des_{app}, r`)`
    if $(v = SubOptimalBinding)$ **then**
        $Ref_{app} \leftarrow$ `insertBefore(`Des_{app}, U_r, X_r`)`
        $Ref_{app} \leftarrow$ `insertAfter(`Ref_{app}, U_r, Y_r`)`
    end if
    if $(v = NestedUsage)$ **then**
        $Ref_{app} \leftarrow$`merge(`Des_{app}, X_r`)`
        $Ref_{app} \leftarrow$`merge(`Ref_{app}, Y_r`)`
    end if
    if $(v = TradeOff)$ **then**
        $Ref_{app} \leftarrow$ `reconfigure(`$Des_{app}, AppCt, X_r$`)`
    end if
    if $(v = ResourceLeak)$ **then**
        $Ref_{app} \leftarrow$
        `insertAfter(`Des_{app}, U_r`, {getRelSysCall(`r`)})`
    end if
    $\langle v, r \rangle \leftarrow$ `getDefect(`$Res_{app}, App, AppCt$`)`
end while

7.4 EVALUATION

In this section we shall describe the experimental setup and the subject apps (in Section 7.4.1) and key results of the evaluation (in Section 7.4.2). Finally, we will present a case study of one of the subject apps, *Sensorium*, in Section 7.4.3.

7.4.1 Subject Apps and Experimental Setup

Primarily, we wish to evaluate the efficacy of our technique in detecting the presence of inefficient design-patterns (*cf.* Definition 7.2.1) and in generating usable energy-efficient re-factoring of the aforementioned patterns, in real-life apps. To achieve this objective, we create a suite of subject-apps, consisting of ten, open-source application from the F-droid, open-source Android app repository. These apps [108, 172, 162, 169, 173, 171, 170, 166, 168, 167] are diverse in terms of functionality and size (*cf.* Table 14), thereby allowing us to evaluate the different aspects of our framework. It is worthwhile to know that significant energy-inefficiencies can manifest only in those apps where energy-intensive Android API calls have been used (as discussed in Section 7.3), therefore, while creating the suite of subject apps we only choose apps that have usage of atleast one energy-intensive component (such as wakelock, GPS, Wifi, etc). In addition, we also try to make the suite of subject apps diverse by including apps that use different kinds of energy-intensive resources (such as wakelock, sensor, GPS, Wifi, etc). Our re-factoring framework and a power measurement utility, were run on a Desktop PC. The Desktop-PC was equipped with an Intel i7 processor, 8 GB main memory and Windows 7 OS.

Measuring energy consumption: To measure the energy consumption of the mobile device, we created an experimental setup as shown in Figure 57(a). We used the Monsoon Power Monitor [175] to supply the mobile device with a steady voltage of 4.2 Volts and to measure its power consumption. The mobile device used in our experiments was Samsung S4, running an Android KitKat OS (version 4.4.2). While the app was executed on the mobile device, we continuously measure the power consumption of the mobile device using the Monsoon Power Monitor which can sample at 5KHz. To maintain consistency in power measurements across our experiments, we have followed a few timing restriction in providing test-inputs to the app (as shown in Figure 57(b)). For instance, the interval between two input-events (such as touches,taps,clicks) was 15 seconds. Additionally, an idle time (of 45 seconds), was observed just after the app had started or stopped execution. Finally, the screen time-out duration of the mobile device was set to 15 seconds. The inputs (to the app) were encoded as monkeyrunner scripts and were invoked from the Desktop PC.

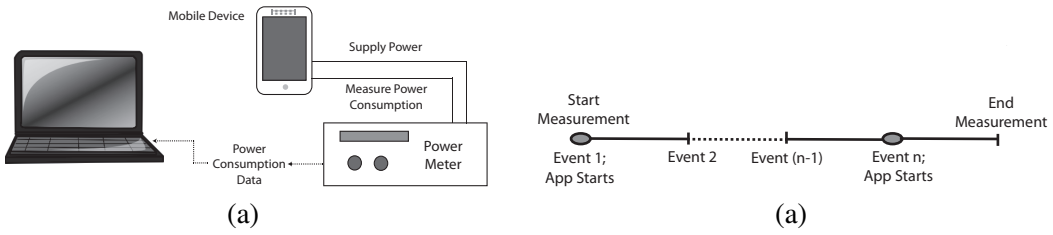


Figure 57: (a) Measurement setup (b) Timing parameters

7.4.2 Key Results

Of the ten apps studied in the evaluation, we found sub-optimal resource bindings in three apps, nested resource usage in one app, QoS trade-offs in six apps and resource leaks in five apps. Even though these apps are of considerable size (for instance, app *Sensorium* has $4,001$ LoC, apk size is $1,248$ KB), our framework was able to generate design expressions in less than a minute's time. This goes on to show that our technique can be scalably applied to real-world apps. It is worthwhile to know that the design expression generation time excludes the time required for EFG generation, which is done with the help of a third-party tool Dynodroid. The EFG can be generated offline and needs to be updated only if the GUI layout of the app is updated. Currently our framework can not only, produce the re-factored expression but also generate file/class names and method names, as well as lines numbers, where the re-factoring must be done. When we applied the re-factoring suggested by our framework, we observed a reduction in energy-consumption between 3 % to 29 %(Table 14).

7.4.3 Case Study

Sensorium is a publicly available Android app which allows its user to collect sensor data such as network signal strength, location information, battery status, etc. We specifically choose this app out of the ten subject apps used in our evaluation because of its long and well maintained repository at GitHub [176]. This project was active for a period of approximately two years, in which duration it saw 214 commits. However, due to space restrictions, it would be impractical to discuss the design (& its re-factorings) for all of these 214 commits. Therefore, we choose only 6 important commits (here referred as commits a to f), by observing a plot of changes (both GUI, as well as, source code), as shown in Figure 58.

$$\text{Original Expression} \quad : \quad G_1 G_2 S E \tag{50}$$
$$\text{Re-factored Expression } (i) \quad : \quad G_1 G_2 S G_1' G_2' \tag{51}$$
$$\text{Re-factored Expression } (ii) \quad : \quad G_1 S G_1' \tag{52}$$

During the earlier stages of the project (*cf.* commits $a - c$), the code and layout are changed heavily across consecutive commits (*cf.* Table 15) Whereas in the later commits (*i.e.* commits $d - f$), where the project is fairly mature and stable, the GUI layout and the design does not change substantially. During the evolution of the project, a number of commits had one or more re-factoring opportunities due to *sub-optimal binding* (due to sub-optimal sensor acquisitions), *nested usage* (such as nested location updates in commit b) and *resource leaks* (due to not releasing the sensor on app exit). These defects were successfully detected and re-factorings suggested by our framework. For instance, commit b (design expression shown in expression 50, constituent symbols described in Table 15), could be re-factored for resource leaks, such as shown in expressions 51 and nested usage, such as shown in expression 52.

7.5 COMPARISON WITH EXISTING WORKS

In this section we shall discuss some existing works, specifically related to (i) Understanding energy-inefficient behaviour, (ii) Detecting energy-inefficient behaviour and (iii) Optimizing energy-consumption behaviour.

Understanding energy-inefficient behaviour: The first step in resolving energy-inefficient behaviour is to understand its characteristics. Recent works such as [177, 79, 3, 72, 73, 74, 178, 179, 180, 181, 182, 183, 184, 185, 186] have presented some interesting insights towards understanding energy-inefficient behaviour in mobile apps. Profiling work such as [3] present insights such as the fact that I/O components (such as GPS, Wifi) and power management utilities (such as Android Wakelocks) are usually responsible for high energy consumption on mobile apps. Works such [177, 73, 74] have presented frameworks that use energy-models to estimate the energy consumption of an app, for a given workload. In particular, the works of [74] and [73] present techniques to map the energy-profile of an app (for a given input), to the app source-code. However, a key limitation of profiling-based techniques is that they heavily depend on test-inputs to generate the energy-profile. Manually, obtaining suitable test-inputs that can expose energy-inefficient behaviour is often non-trivial. In comparison, in our framework we use design expressions (more specifically, intersection between design expressions and defect expressions) to detect energy-inefficiencies in an app.

Detecting energy-inefficient behaviour: In recent years, a number of works [76, 77, 187, 83, 178, 188, 189, 190] have proposed dynamic, as well as static program analysis techniques for detecting energy-inefficient behaviour in mobile apps. Dynamic program analysis techniques such as [79], use symbolic execution to estimate the energy consumption for a given path in a program. Other works [76, 77], have used static program analysis techniques to detect the presence of resource leaks in apps. Test-generation techniques for mobile apps have been mostly applied to functional properties. However, a few works, such as [187, 83], exist that assist in energy-aware test-generation. Our recent work [191] presents a framework that uses energy-inefficient design patterns to debug and localize field failures in mobile-apps. In general, techniques described in this paragraph can assist in detecting energy-inefficiencies in an app post-development, however, such techniques do not provide support for energy-aware app re-factoring. In contrast, our technique is specifically designed to assist the app-developer by suggesting energy-efficient re-factorings, during the app-development stage.

Table 15: Design expression and re-factorings for commits highlighted in Figure 58

Commit Hash (#)	Code Layout Changes	Expression Expression	Re-factored Design Expression	Comments
73b0444 (a)	Initial commit. Event handler not attached to events.	SE	SE	n/a
9d141b7 (b)	Add more sensors (location)	G_1G_2SE	$G_1SG_1{}'E$	Resource Leaks; Nested Usage
94c58c3 (c)	Changed layout, removed files	$G_1G_2(((SM)\|SDD_1)\|$ $SCC_1(G_1G_2\|G_1'G_2'))E\|$ $G_1G_2SDD_1E\|G_1G_2S$ $CC_1(G_1G_2\|G_1'G_2')T^*E$	$(((G_1SG_1'M)\|G_1$ $SG_1'G_1DD_1G_1')\|G_1$ $SG_1'CC_1)E\|G_1SG_1'$ $G_1DD_1G_1'E\|G_1$ $SG_1'CC_1T^*E$	Sub-optimal resource binding; Resource leaks; Nested Usage
e2aebfe (d)	Change layout of main screen	$G_1G_2S_1((((S\|SM^+)\|$ $SM^+CC_1G_1G_2S_1)\|$ $SM^+DD_1)\|SM^+C_1G_1'$ $G_2'S_1{}'CC_1)E$	$((((G_1S_1SG_1'S_1'\|$ $G_1S_1SG_1'S_1'M^+)\|$ $G_1S_1SG_1'S_1'M^+$ $CC_1)\|G_1S_1SG_1'S_1'M^+$ $G_1S_1DD_1G_1'S_1')\|G_1$ $S_1SG_1'S_1'M^+(CC_1))E$	Sub-optimal resource binding; Resource leaks; Nested Usage
fd643d7 (e)	Added more sensor (pressure)			
832aa14 (f)	Most recent commit			

E - Exit app
T - Summary
M - Menu button pressed
S - Main activity on screen
D - Debug activity
D_1 - Debug activity, (sub event 1)
C - Configuration activity
G_1 - Location acquire, first occurrence
G_2 - Location acquire, second occurrence
S_1 - Pressure sensor acquire, first occurrence
$G_1{}'$ - Release of resource G_1
$G_2{}'$ - Release of resource G_2
$S_1{}'$ - Release of resource S_1
C_1 - Configuration activity, (sub event 1)

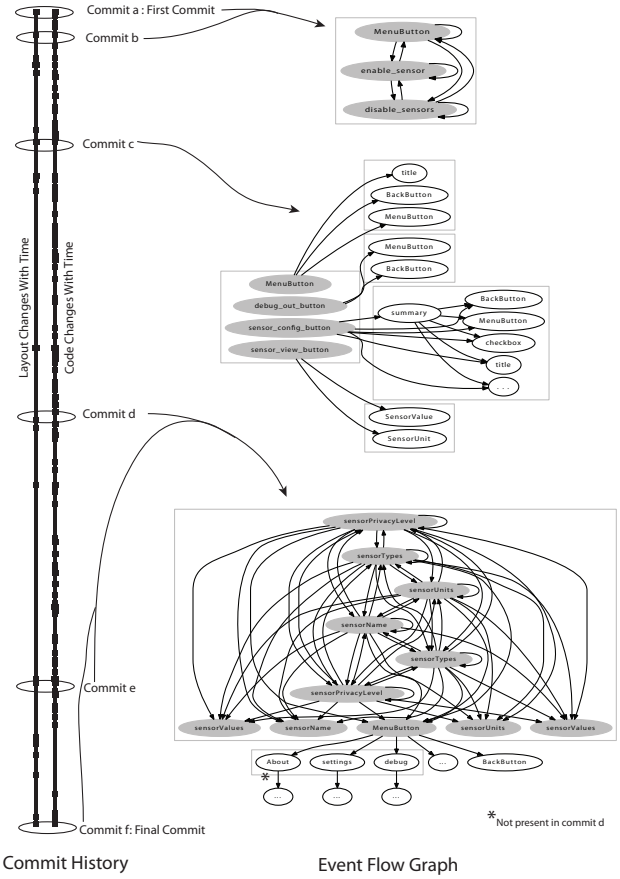


Figure 58: Some commit from the 214 commits of the project *Sensorium*

Optimizing energy-consumption behaviour: A number of orthogonal approaches [86, 84, 192, 193]have been presented over the recent year to optimize energy-efficiency of programs. For instance, [86] proposes the use of a new energy-aware programming language. Such languages, if used, can be instrumental in developing energy-efficient application, however, so far, such languages have not witnessed widespread use. Another group of work [84, 192] focuses on using energy-aware optimization. [84] in particular proposes the use of *approximate implementations* [84]. The key idea behind this work is to encode multiple, approximate implementations of a given (time-consuming) computation, such as loops. Since in mobile apps time-consuming computation may not necessarily imply energy intensive computation (because CPU may have a lesser power consumption than I/O components [187]), therefore, direct use of *approximate implementation* may not very beneficial for mobile-apps. However, the underlying philosophy of trade-offs between QoS and energy-efficiency is useful and therefore adapted to our technique as well. One of our previous articles [193], discusses the potential for energy-aware programming, however, it does not provide a framework necessary to conduct energy-aware re-factoring. Another preliminary work [194] proposes a re-factoring technique that uses compiler optimization to improve energy-efficiency of *Observer* and *Decorator* design patterns in object-oriented programs.

A different line of work leverages the power of genetic algorithm (GA) to search for energy-efficient versions of a given program. The key idea in such works [195, 196] is to iteratively search for mutated program versions that have the same functional behaviour as the original program version (program behaviour compared using existing test-cases), but are more energy-efficient. Mutated versions of a program can be obtained by applying mutation operators such as *copy*, *swap*, *delete*, etc, on intermediate representation [195] of program or even assemblies [196]. A key difference between our re-factoring technique and such GA based energy-efficiency optimization techniques is that these (GA based) optimization techniques are often well suited for post-development optimization phases, more so in scenarios where the developer does not have necessary knowledge or intent of creating designs that are specifically energy-efficient. On the contrary, our approach actively highlights design in-efficiencies to the developer and produces re-factorings that may lead to significant changes in the design of the program and is therefore more suitable for use in early stage of the software development life-cycle.

7.6 THREATS TO VALIDITY

A threat to the validity of our framework may arise due to the incompleteness of the EFG model. The dynamic exploration technique (to create EFG), in our approach may not able to generate a complete UI model (EFG) for a given app. This may cause certain part of the app code to be unmodeled and hence unanalysed by our framework. It is worthwhile to mention alternative static analysis based techniques (such as [77]) that are based on parsing of XML-based UI files, may also be unable to generate a complete UI model for a given app because Android framework allows creation of dynamic UI screen programmatically. To the best our knowledge no existing work provides a technique for complete EFG generation. However, since the design expression generation part and the re-factoring part of our framework are loosely coupled with the EFG generation part, if any complete EFG generation technique is developed in future we can easily integrate it with our framework. Another threat to validity to this work may arise due to the choice of subject programs. Since we needed open source apps of our experiments we were restricted to Fdroid open-source app repository. Even though Fdroid is the biggest app repository of its kind, it is still small as compared to Google Play Store. This may have introduced some sampling bias [197] in our results.

7.7 CHAPTER SUMMARY

In this chapter, we present a technique to address the need for tools that can assist in energy-aware app development. Our technique uses a set of energy-efficiency guidelines to re-factor the design expression of an app. A design-expression is a regular-expression that represents the ordering of energy-intensive, resource usages and invocation of key functionalities (event-handlers) within the app. As a result of using design-expressions, our re-factored technique is not limited by event-handler (class/method) boundaries. This not only increases the re-factoring opportunities but also makes our technique scalable. To demonstrate the efficacy of our technique we analysed a suite of open-source, apps with our technique. The resultant re-factoring when applied, reduced the energy-consumption of these apps between 3 % to 29 %. We also present a case study for one of our subject apps that captures its design evolution over a period of two-years and more than 200 commits. Our framework found re-factoring opportunities in a number of commits, that could have been implemented earlier on in the development stages, had the developer used an energy-aware re-factoring technique such as the one presented in this work.

8 DEBUGGING ENERGY-EFFICIENCY RELATED FIELD-FAILURES IN MOBILE-APPS

Debugging *field failures* can be a challenging task for app-developers. Insufficient or unreliable information, improper assumptions and multitude of devices (smartphones) being used, are just some of the many factors that may contribute to its challenges. In this work, we design and develop an *open-source framework that helps to communicate, localize and patch energy consumption related field failures in Android apps*. Our framework consists of two sets of *automated tools*: one for the app-user to precisely record and report *field failures* observed in real-life apps, and the other assists the developer by automatically localizing the reported defects and suggesting patch locations. More specifically, the tools on the developer's side consist of an Eclipse-plugin that detects specific patterns of Android API calls, that are indicative of energy-inefficient behaviour. In our experiments with real-life apps we observed that our framework can localize defects in a short amount of time (~3 seconds), even for apps with thousands of lines-of-code. Additionally, the energy savings generated as a result of the patched defects are significant (observed energy savings of up to 29%). When comparing the patch locations suggested by our framework to the changes in the patched code from real-life app-repositories, we observed a significant correlation (changes suggested by our tool also appeared in the source-code commits where the reported defects were marked as fixed).

8.1 INTRODUCTION

Debugging *field failures*, even for functionality related defects, can be a challenging process for the developers [198, 199, 200, 201]. It can be even more challenging in the case of non-functional defects (such as energy-inefficiencies) because such defects depend not only on the failure-revealing inputs but also on the state of the hardware device on which the app is executed. Online coding repositories often provide tools such as issue tracking systems, discussion forums, etc, to alleviate this problem, however, in many scenarios such tools are insufficient. Consider the example of Issue 520 (*Battery drain when not in use*) for the app *MyTracks* [202] (more than 10 Million downloads on Google Play Store). Since the original commenter had reported this issue, a total of 43 people (including project members) have participated in the discussion over a period of 4 years to debug the issue. Participants of the discussion have provided test-cases and device descriptions, and exchanged various versions of app files, snapshots, log files, changed devices, etc. However, these ad-hoc methods have done anything but to further confuse the involved parties. Here is an excerpt from their conversation.

Comment 1 Jul 21, 2011	When I don't use MyTracks, I don't expect to see it at the top of the list of applications draining the battery
Comment 3 Jul 23, 2011	I don't really understand this issue. I checked the log but I did not see any suspicious. . . .
Comment 5 Jul 27, 2011	Answer to comment 3: if you examine the log related to comment 2, you will see that MyTracks has never been launched but the application still won the third prize for power consumption.
Comment 35 Oct 3, 2012	. . . Using a battery monitor called "GSam Battery Monitor", it give more information about what "My Tracks" is using: "Orientation", and it's the only things use by "My Tracks" when it does not track anything (no CPU, no wakelock ..)
Comment 40 Jul 13, 2013	In response to comment 35, we have removed the "Orientation" sensor in My Tracks version 2.0.5
Comment 41 Jul 13, 2013	I need help to debug this further. . . . wondering . . . I can email you a few APKs to try.

This example goes on to show how challenging it could be to communicate and debug defects in real-world apps even when both the parties (the user who reports the defect and the developer) were willing to cooperate. As highlighted in our previous work [159], a mobile app may demonstrate energy-inefficient behaviour due to a number of reasons (*cf* Table 17). Some of these defects may only manifest when complex sequence of interactions (or patterns) occur during the execution of the app. For example, suboptimal resource binding (binding resource too-early or unbinding too late) may make an app energy-inefficient. Another example could be the scattered usage of network components that cause power loss due to *Tail Energy*[203]. To localise such energy-consumption related defects, the developer needs to track relevant (energy/computation intensive) Android API calls in a context-sensitive manner (*where*, *when*, and *how* within the activity life-cycle). However, such contextual debugging information is much beyond user's capability to collect for communication. As a result, often the information exchanged between the two parties is insufficient, vague and sometimes even misleading. *What is needed to solve this problem is a framework that can provide a reliable yet succinct means of communicating user-observed defects and an automated technique to use this communicated information to localize and debug these defects on the developer side.*

In this chapter we present a framework that can break the communication barriers between users and developers on assisting in energy-aware debugging. Figure 59 presents an overview of our framework, which contains two main parts: one for the user and the other for the developer. The user side tools consist of an instrumentation utility, which can automatically instrument a given apk file, followed by a customized logging utility, which records the log messages and system states at runtime. To report a defect the user simply executes the instrumented app, with the failure-revealing test inputs, on her mobile device, while the logging utility is running. The log file recorded automatically thus contains all the information that is needed to localize/patch the defect and can be sent to the developer.

The developer side tool consists of an Eclipse plugin, in which the debugging framework has been embedded. Provided with the log file, the debugging framework generates a profile call graph, which is subsequently analysed through a contextual analyzer to identify Android API call patterns for energy-inefficiencies. The pattern-based fault-localization technique is motivated by the fault-model presented in [159]. If such patterns are found, the framework localizes the defect in the app source-code and also suggests potential patch locations. It also provides a visual representation of observed defect. The framework on the developer side works in an automated fashion and requires minimal configuration.

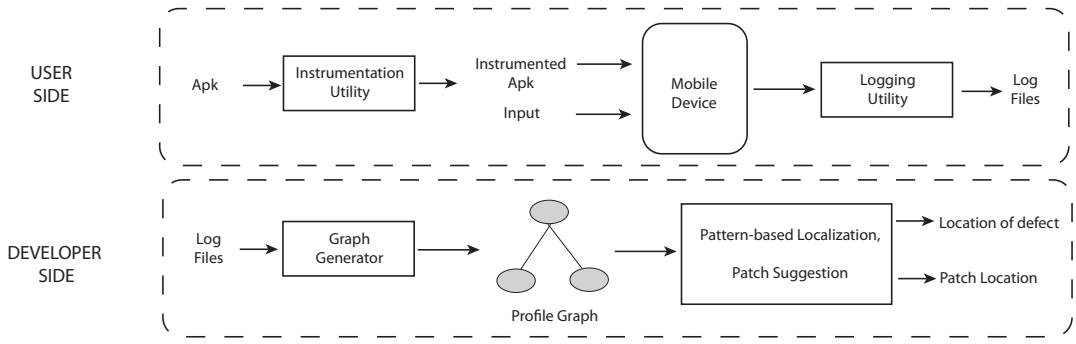


Figure 59: Overview of log-based, energy-inefficiency localization in mobile apps

8.2 DETAILED METHODOLOGY

In this section we shall discuss the (i) Instrumentation & Logging, (ii) Profile Graph Generation, (iii) Pattern-based contextual analysis for energy-inefficient behaviour detection and (iv) Defect localization and patch suggestion.

8.2.1 Instrumentation and Logging

Before the app can be executed to generate the log files, it has to be instrumented by our tool. Table 16 lists some of the packages from which methods are instrumented by our tool. These methods can be divided into two categories (i) event-handlers, that are invoked on arrival of events and (ii) Android API call that may have significant impact on the energy consumption. For example, the package `android.app.activity` (in the category event handlers), houses the method that are called when an activity is created (onCreate) or when an activity is paused (onPause), etc. Whereas, the package `android.hardware.*` (in the category Android API calls), houses the methods that are used to access hardware components such as `Sensors` and `Cameras`.

```
1  public void onCreate(Bundle paramBundle)
2  {
3    Log.i("EventHandler","PackageName\ActivityName\onCreate"+Process.myTid());
4    //...
5    Log.v("AndroidAPI", "Landroid/hardware/Camera;->open()");
6    Camera localCamera = Camera.open();
7    //...
8    Log.i("Parameters", "Landroid/hardware/Camera$Parameters;->setFlashMode" + "off");
9    Log.v("AndroidAPI", "Landroid/hardware/Camera$Parameters;->setFlashMode");
10   parameters.setFlashMode("off");
11   Log.i("EventHandler", "Finished_onCreate");
12 }
```

Figure 60: Example of code instrumentation

Figure 60 shows an example code after instrumentation. Observe that for the event-handler onCreate both `start` and `end` have been instrumented (lines 3 and 11, respectively). Additionally, the log messages contain the thread id which helps in resolving ambiguity in presence of multiple threads. All Android API calls which belong to packages described in Table

Table 16: Event-handlers and Android API calls that are instrumented

Category	Type	Package Name
Event Handlers	Activity	`android.app.activity`
	Service	`android.app.service`
	Receiver	`android.content.BroadcastReceiver`
Android API calls	Bluetooth	`android.bluetooth.*`
	Sensors	`android.hardware.*`
	Location	`android.location.*`
	Multimedia	`android.media.*`
	PowerManager	`android.os.PowerManager.*`
	Database	`android.content.SQLite.*`
	SMS	`android.telephony.SmsManager`
	Telephony	`android.telephony.TelephonyManager`
	Network (A)	`android.net.*`
	Network (J)	`java.net.URL`
	IO	`java.io.*`
	Cipher	`javax.crypto.Cipher`
	Apache Http	`org.apache.http.*`
	Thread	`java.lang.thread`

16 are instrumented in a similar manner (lines 5 and 6). For certain Android API call we need to add an additional line of instrumentation to record their parameters. This because for such Android API calls, their parameters may decide the resultant power consumption. For instance, in Figure 60, the parameters to the API call `setFlashMode` decide the state of flash (off, on, torch, etc) post invocation. Therefore, an additional line (*cf.* line 8) has been instrumented.

It is worthwhile to know the instrumentation tool works on apk file (source files not needed) and is very light weight, hence it can be run on any commodity machine on the user side. Also it is relatively easy to use as it does not require any prior configuration. Once the target apk files have been instrumented, the app-user (who wishes to report the defect) can execute the instrumented app on her mobile device. This execution should be done while our logging utility is monitoring over android debug bridge (adb). To do so, the user simply needs to connect the mobile-device through a USB cable to the PC on which the logging utility is running. The logging utility specifically monitors and records the log messages that are instrumented by the logging utility. In addition, it also records information relevant to device's power consumption, such as status of wakelocks, frequency of GPS updates, etc.

8.2.2 Profile Graph Generation

The energy profile graph for an Android app is a dynamic call graph [204], that represents call dependencies between various subroutines that are executed in a given execution of the app. It primarily emphasizes on the method invocations related to energy consumption and app's GUI *activity* life-cycle. Call graph profiling [204, 205, 206, 207] has been an effective method of program analysis and has been used in applications such as performance analysis and compiler optimizations. For example, in context of performance analysis, call-graph profiling has been used to monitor per-subroutine time consumption, based on which performance bottlenecks can be identified. The energy profile graph is based on similar principles but it is bit more sophisticated than the standard call-graph. Not only does it record the amount of time spent within each application component, but also records the behaviour of energy-

```
 1 12:31:22.471: DummyEvent: AndroidApp()
 2 12:31:22.471: EventHandler(240049): com\totsp\crossword\shortyz\ShortyzApplication_onCreate()
 3 ...
 4 12:31:22.481: EventHandler(240049): Finished_onCreate
 5 12:31:22.481: EventHandler(240049): com\totsp\crossword\BrowserActivity_<init>()
 6 ...
 7 12:31:22.486: EventHandler(240049): Finished_<init>
 8 12:31:22.496: EventHandler(240049): com\totsp\crossword\BrowserActivity_onCreate()
 9 ...
10 12:31:22.566: EventHandler(240049): Finished_onCreate
11 12:31:22.566: EventHandler(240049): com\totsp\crossword\BrowserActivity_onResume()
12 ...
13 12:31:22.571: EventHandler(240049): Finished_onResume
14 ...
15 12:31:56.136: EventHandler(240049): com\totsp\crossword\BrowserActivity$5_run()
16 12:31:56.136: EventHandler(240049): com\totsp\crossword\net\Downloaders_download()
17 12:31:56.161: EventHandler(240049): com\totsp\crossword\net\ThinksDownloader_download()
18 12:31:56.171: EventHandler(240049): java/net/URL;-><init>()
19 12:31:56.231: EventHandler(240049):org/apache/http/client/methods/HttpGet;-><init>()
20 12:31:59.626: EventHandler(240049): com\totsp\crossword\net\AbstractDownloader_copySream()
21 ...
22 12:31:59.966: EventHandler(240049): Finished_copyStream()
23 12:31:59.966: EventHandler(240049): Finished_download(Ljava/util/Date;)Ljava/io/File;
24 12:31:59.971: EventHandler(240049): com\totsp\crossword\net\Downloaders_processDownloadedPuzzle()
25 12:32:00.861: EventHandler(240049):  Finished_processDownloadedPuzzle
26 12:32:06.526: EventHandler(240049): Finished_download(Ljava/util/Date;Ljava/util/List;)V
27 ...
28 12:32:07.801: EventHandler(240049): Finished_run
29 ...
30 12:32:31.786: DummyEvent: Finished_AndroidApp
```

Figure 61: Log-messages generated from Shortyz app

intensive resources (such as the GPS, Screen, Wifi, etc) and status of activity life-cycle within a given application. It is worthwhile to note that such a call graph is possible because of instrumentation as described in section 8.2.1.

The log (generated using the logging utility, *cf.* Figure 59) is a well-structured, hierarchical text file. The entries in the log file are ordered by time-stamps. Figure 61 shows an example of such a log file. Note that since an Android app is event-driven, it does not have an explicit `main` method. Therefore, for the purpose of clarity we introduce a pair of dummy events, labelled `AndroidApp()` and `Finished_AndroidApp()`, that represents the start and end of execution of the app. Such a pair of entry-exit events forms the basis of the nested hierarchical structure for the call graph, since for each event-handler e, all of its subroutines, including method-invocations, are logged within the (entry-exit) pair of events for e. In Figure 61, such an entry-exit pair can be found on lines 15 and 28, respectively, and examples of method-invocation can be found on lines 18 and 19.

During the log-file parsing (to generate the profile call graph), we also create two axillary data structures: *AndroidActivityTable* and *ResourceActivityTable*. As the name suggests, the *AndroidActivityTable* keeps track of activity related nodes (such as `onCreate`, `onResume`) in the profile call graph, and *ResourceActivityTable* keeps track of nodes related to energy-intensive resources. The actual processing of the log-file happens in two passes:

First Pass: For each event-handler, entry-exit events are matched. Method-invocations that are related to energy-intensive resources are recorded in the *ResourceActivityTable*. Activity stages are recorded in *AndroidActivityTable*. Two implementation issues that may arise during this pass are as follows:

Abnormal Exits	Abnormal exits (from method calls) may happen due to runtime exceptions. This may create unmatched method call (entry-exit) pairs in the log file. To detect such issues, we maintain an environment stack. When an exit event does not match with the top entry event in the environment stack, it implies that the method call corresponding to the top entry event, may have had an abnormal exit.
Multi-Threading	Log messages from multiple simultaneously executing threads (within the app) can also disrupt the balanced entry-exit pairs in the log file. To counter this issue, our runtime logging utility records the thread ID for each logged event. This allows us to do the entry-exit pair matching for each thread.

Second Pass: Well-balanced, (entry-exit) paired events are used to construct a hierarchical call-dependency graph, where each node contains the following fields supporting the first-child/next-sibling representation:

```
class Node {
  Node firstChild;   // first subroutine child
  Node nextSibling;  // next sibling subroutine
  Node parent;       // parent caller
  String className;  // class for current method
  String methodName; // current method name
  long timeStamp;    // logging timeStamp
  long runtime;      // runtime between entry-exit events
  boolean MPCC;      // if method contains Android API calls
  ...                // of interest
}
```

Figure 62 shows a partial call graph for one of the analysed apps, where rectangular nodes are used to represent event-handler nodes and round nodes are used to represent Android API calls.

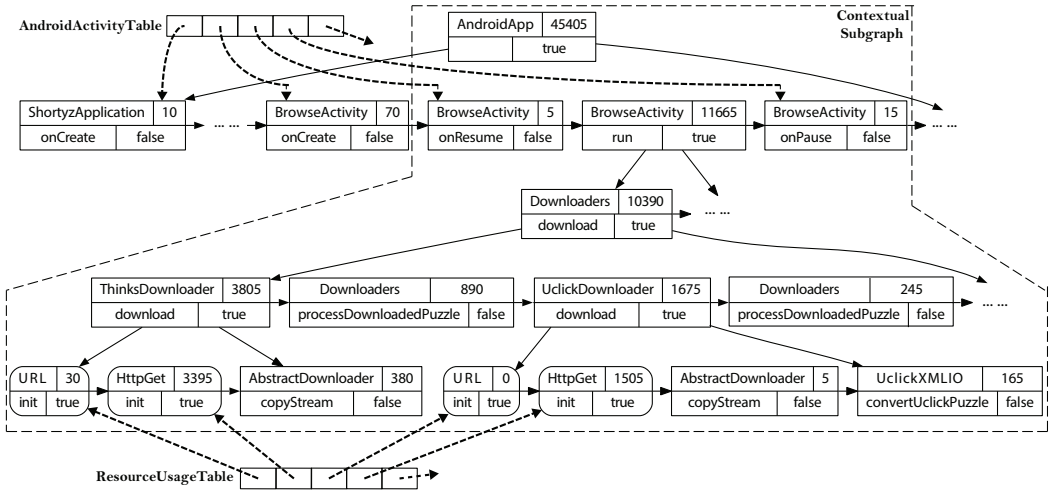


Figure 62: Partial profile call graph for Shortyz app

Table 17: List of energy-inefficiency related defects with defect pattern, patch suggestion, affected hardware components and a real-world example with user comments.

	Defect Type	Affected Components	Defect Pattern (P)	Patch Suggestion	Real-world example
Hardware Resources	Resource Leaks	I/O Components $i \in$ {Wifi, Sensor, GPS, Camera }	$P_{\text{resource leak}} \rightarrow a_i\ Q$ $Q \rightarrow X_{\bar{i}} \mid X_{\bar{i}}\ u_i\ Q$ $X_{\bar{i}} \rightarrow \epsilon \mid x_{\bar{i}}\ X_{\bar{i}}$	Release a_i after PL($P_{\text{resource leak}}$)	Csipsimple Issue 81 [208] "When I run Sipdroid, by the end of the day I have battery indicator at about 50%. When I run CSipSimple instead, with all other activities basically the same, by end of the day the battery indicator is yellow or even red (< 20% I think)."
	Suboptimal Resource Binding		$P_{\text{suboptimal binding}} \rightarrow a_i\ Q\ r_i$ $Q \rightarrow X_{\bar{i}}\ u_i\ X_{\bar{i}} \mid X_{\bar{i}}\ u_i\ Q$ $X_{\bar{i}} \rightarrow \epsilon \mid x_{\bar{i}}\ X_{\bar{i}}$	Re-factor $X_{\bar{i}}$	Sofia Public Transport Issue 38 [209] "Disable GPS updating when estimates dialogue is displayed . . . This will " reduce battery usage
Sleep-state Transition Heuristics	Wakelock Bug	Power Management $p \in$ {CPU, Screen, Keypad} $c \in$ {CPU}	$P_{\text{wakelock bug}} \rightarrow a_p\ Q\ X_{\bar{p}}$ $Q \rightarrow X_{\bar{p}} \mid a_p\ Q\ r_p \mid Q\ Q$ $X_{\bar{p}} \rightarrow \epsilon \mid x_{\bar{p}}\ X_{\bar{p}}$	Add r_p to match the reference count of a_p	Adw Launcher Android Issue 202 [210] "This could lead to some battery drain, or with our AMOLED screens - burn in. I discovered this by accidentally hitting the app drawer button when setting my phone on my desk, I assumed it would turn it self off but after 15 minutes never did."
	Tail-energy Hotspot	Network $n \in$ { Wifi, 4G, 3G,2G }	$P_{\text{tail energy}} \rightarrow a_n\ Q\ r_n$ $Q \rightarrow X_c\ u_n\ X_c \mid X_c\ u_n\ Q$ $X_c \rightarrow \epsilon \mid x_c\ X_c$	Re-factor X_c	Google Voice Location Issue 4 [211] "Feature Suggestion: Optimize Data Usage . . . Totally agreed. I've used this app on three different phones, and I've seen noticeable battery drainage when it's active."
Background Services	Vacuous Services	Background Services $s \in$ {GPS, Sensors, Wifi,4G,3G,2G }	$P_{\text{vacuous services}} \rightarrow a_s\ Q$ $Q \rightarrow X_{\bar{s}} \mid X_s\ u_s\ Q$ $X_{\bar{s}} \rightarrow \epsilon \mid x_{\bar{s}}\ X_{\bar{s}}$	Stop service a_s after PL($P_{\text{vacuous services}}$)	Recycle-locator Issue 33 [149] "Battery life is absolutely terrible . . .It appears to be due to the GPS constantly searching for a signal."
	Expensive Services		$P_{\text{expensive services}} \rightarrow a_s\ u_s\ r_s$ If configuration parameters of a_s do not follow energy-efficiency guidelines [213]	Change configuration parameters for a_s	Osmdroid Issue 76 [212] "MyLocationOverlay class uses 0 as default values for requesting location updates. 0 (fastest updates possible) is maybe an overkill and drains some battery" . . .
Defective Functionality	Immortality Bug	All of the above	$P_{\text{immortality bug}} \equiv \{$ LaunchonReboot (PL ($P_{\text{resource leak}}$)) $\vee$ LaunchonReboot(PL ($P_{\text{wakelock bug}}$)) $\vee$ LaunchonReboot(PL ($P_{\text{vacuous services}}$)) $\}$	Use patch suggestion for respective issue. May re-factor PL($< pattern >$) from methods that launch on reboot	Omnidroid Issue 98 [214] "Battery Drain Concerns . . . I know it was omnidroid because I looked at the battery use graph and omnidroid was using 8 times more battery than the next highest app. It dwarfed everything else." (app restarts itself on reboot)
	Loop-energy Hotspot		$P_{\text{loop hotspot}} \equiv \{$ InLoop(PL ($P_{\text{tail energy}}$)) $\vee$ InLoop(PL ($P_{\text{suboptimal binding}}$)) $\vee$ InLoop(PL ($P_{\text{expensive services}}$)) $\}$	Use patch suggestion for respective issue. May re-factor PL($< pattern >$) out of loops	K9Mail Issue 424 [215] "I found today that when my mail accounts were temporarily unreachable due to a server glitch . . . battery drained *very* quickly while the phone became noticeably warmer even though it was in standby mode with the screen display off. Also, k9mail is AFAIK the only third-party background service that I have running"

a : Android API call to acquires a I/O component(h)/power management utility(p)[1]/services(s)[2]
u : Android API call that uses a I/O component(h)/power management utility(p)/service(s)
r : Android API call to release a I/O component(h)/power management utility(p)/service(s)
x : any Android API call

$PL : P_{defectcategory} \rightarrow location$
$LaunchonReboot : location \rightarrow boolean$
$InLoop : location \rightarrow boolean$

1 : power management utilities can be reference counted
2 : of the many possible services, only the energy-intensive services, such as the once using sensors, are monitored

8.2.3 Patterns for Energy-inefficient Behaviour

Table 17 presents a list of energy-inefficiency related defects commonly observed in mobile apps. For each defect category, we present the pattern of Android API calls that indicate energy-inefficient behaviour, the set of affected components and a real-world example along with user comments, where such a defect has been observed. These defect patterns are derived from the energy-inefficiency fault-model presented in our previous work [159].

Most of the patterns ($P_{defectcategory}$) in Table 17 are represented using a context-free grammar. The symbols $a_{component}$, $u_{component}$ and $r_{component}$ are used to denote the Android API calls for acquire, usage and release of a component. Whereas, the symbol $x_{\overline{component}}$ is used to denote all Android API calls *not* associated with a component. The symbols ϵ, a, u, r, x are used as terminals, whereas the symbols Q and X are used as variables. Rest of symbols used in the patterns retain their conventional meaning.

The patterns for $P_{\text{immortality bug}}$ and $P_{\text{loop hotspot}}$ require some additional explanation as they use non-standard notations. $P_{\text{immortality bug}}$, in particular, is a disjunction of three separate scenarios. Specifically, if patterns consistent with $P_{\text{resource leaks}}$, $P_{\text{wakelock bugs}}$ or $P_{\text{vacuous services}}$ are detected and localized to a program location which is restarted every time the system (mobile device) is rebooted, then our framework reports the presence of an immortality bug. For this defect category, we define two additional functions *PL* and *LaunchonReboot*. The function, $PL : P_{defectcategory} \rightarrow location$, provides the program location ($\langle class, method, lineno \rangle$) for a given pattern (specifically the beginning of the pattern). The function, $LaunchOnReboot : location \rightarrow boolean$, is used to determine the feasibility of a given program location to be launched at reboot. It is worthwhile to know that for Android apps, it is possible to design such a function because the relevant information (which methods will be launched on reboot) is provided statically by means of `AndroidManifest.xml`. Specifically, the function *LaunchOnReboot* keeps a map of all `Receivers` or `Activities` that have registered for the `BOOT_COMPLETED` broadcast intent.

Loop-energy hotspot ($P_{\text{loop hotspot}}$) represents the scenario where an energy hotspot, *i.e.* $P_{\text{suboptimal binding}}$, $P_{\text{tail energy}}$ or $P_{\text{expensive services}}$, is localized to a loop in the app source code. Being executed in a loop can magnify the impact of such defects and therefore may need additional re-factoring effort. An additional function, $InLoop : location \rightarrow boolean$, is added which can tell if a program location falls within the boundaries of a loop construct (*i.e.* `for, while, do-while`). In the following paragraphs, we shall outline the overall algorithm for pattern-detection. However, for the purposes of brevity we shall limit our discussion to only two defect patterns ($P_{\text{resource leaks}}$ and $P_{\text{tail energy}}$).

8.2.4 Contextual Analysis for Energy-inefficient Pattern Detection

The fault-localization process begins by analysing the contextual dependency among Android API calls, activity events, and other work loads, within the profile call graph. Our contextual analysis focuses on the innermost activity cycle for each energy-related Android API call to identify patterns that are associated with energy-inefficient behaviour. The main purpose of our contextual analysis is to extract a contextual subgraph which contains the following information:

WHEN the runtime scenario in which the foreground application activity lies, e.g., `Activity.`*onResume*() and `Activity.`*onPause*(). The *when*-context will help developers to replay the scenario when energy bugs or hotspots happen.

HOW the sequence of Android API call showing how the related hardware resource is acquired, used, and released, if any. The *how*-context is valuable to identifying the faulty patterns and categorizing the type of bugs or hotspots.

WHERE for each Android API call involved in *when* and *how*-context, the method and class details of Android API call, the caller details of Android API call (where it was actually invoked), as well as the subsequent calls to that Android API call are gathered. The subsequent calls are done to resolve ambiguity in case its caller invokes that Android API call multiple times. The *where*-context is critical to fault localization in source code.

Algorithm 4 Energy Defect Detection

```
1:  Global: AT – an abbreviation for AndroidActivityTable
2:          RT – an abbreviation for ResourceUsageTable
3:  Input: resourcesInUse – resources used in the app
4:  Output: a collection of energy defects, if any
5:  function DEFECTDETECTION(resourceInUse)
6:      ds ← ∅
7:      for (each h : resourceInUse) do
8:          for (each defect : h.defectList) do
9:              switch (defect) do
10:               case "resource leaks":                    ▷ Algorithm 5
11:                   ... ...
12:                   break
13:               case "tail-energy hotspot":               ▷ Algorithm 6
14:                   ... ...
15:             end switch
16:         end for
17:     end for
18:     return ds
19: end function
```

Algorithm 4 shows the skeleton of the main procedure, named DEFECTDETECTION, on how to detect all the relevant energy bugs or hotspots given a list of resources in use. For each resource h in use, we prepare $h.defectList$, a list of possible bug or hotspot types associated with h. Our defect detection procedure will perform contextual analysis to investigate the resource usage in an energy profile subgraph to find out whether the runtime trace matches a defect pattern from a list of possible resource-specific defects. Note that `AT` and `RT` are abbreviations for `AndroidActivityTable` and `ResourceUsageTable`, respectively.

Detecting $P_{resource\ leaks}$: Hardware components that are acquired by an app during its execution must be released before exiting, or else, the resources continue to be in high-power state and keep consuming energy. Typically, resources that are acquired during the setup stages of an activity (such as `onResume()`) should be released during its tear-down stages (such as `onPause()`). Failing to follow such protocols may lead to resource leaks. Such scenarios are represented using defect pattern $P_{\text{resource leaks}}$ in Table 17. Algorithm 5 outlines the analysis required to detect such a defect pattern within the contextual dependency subgraph. For each Android API call for resource acquisition (line 2), we first capture its contextual dependency subgraph by locating the foreground running activity while the resource h has been acquired. To determine the temporal boundary of the contextual dependency subgraph, we use the procedures FOREGROUNDBEGIN and FOREGROUNDEND (lines 3-4). These two procedures can be implemented by searching the data-structure AndroidActivityTable for the innermost

Algorithm 5 Contextual Analysis for Resource Leak Bugs

```
1:  for (i ← 0 to RT.size() −1) do                          ▷ each Android API call RT[i]
2:      if (RT[i] ∈ a_h) then                               ▷ Android API call to acquire h
3:          bTime ← FOREGROUNDBEGIN(AT, i)
4:          eTime ← FOREGROUNDEND(AT, i)
5:          j ← i + 1
6:          paired ← false
7:          while (not paired and j < RT.size() and
8:                      RT[j].timeStamp < eTime) do
9:              if (RT[j] ∈ r_h) then                       ▷ Android API call to release h
10:                 paired ← true
11:             end if
12:             j ← j + 1
13:         end while
14:         if (not paired) then
15:             defect.type ← "resource leaks"              ▷ new a defect
16:             defect.info ← {RT[i]}
17:             ds ← ds ∪ {defect}
18:         end if
19:     end if
20: end for
```

activity cycle just wrapping the Android API calls for resource acquisition. Subsequently, our framework checks for a release Android API call for the resource that has been acquired within the subgraph (7-13). Finally, if the release Android API call is not found a defect is recorded (line 14). The data-structure *defect.info* (line 16) maintains a sufficient set of method signatures related to *when*- and *how*-context so that the contextual subgraph can be easily retrieved for fault-localization and defect-visualization.

Detecting $P_{tail\ energy}$: It is common for network components in a mobile device to linger in a high power state, for a short-period of time, after the imposed workload has completed [203]. Such heuristics can reduce the time it takes to transit the device from a low-power (idle) state to an high power (active) state, when the subsequent (network) request arrives. However, while network component waits in a higher-power state, no transmission takes place but additional energy is consumed. The additional energy consumption is referred to as Tail Energy and can be minimized to increase the energy-efficiency of an app. The defect pattern ($P_{\text{tail energy}}$) from Table 17 represents such a scenario. Algorithm 6 shows the contextual analysis required to identify the presence of such a defect pattern. Similar to Algorithm 5, we begin by obtaining the contextual subgraph (lines 4-5). Subsequently, the pattern $P_{\text{tail energy}}$ is searched for in the subgraph (lines 7-14). More specifically, the procedure PROFILEGRAPHBROWSE(RT[i], RT[j]) traverses the profile graph from the node RT[i] to node RT[j], in a temporally-sequential order, (equivalent to pre-order traversal), while collecting CPU intensive processes (X_c) between resource usage Android API calls.

8.2.5 Defect Localization and Patch Suggestion

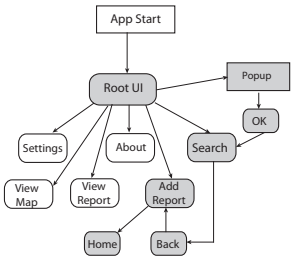
Finding the presence of energy-inefficient patterns is only a part of the debugging effort. The other part involves locating the defect, or rather the origin of defect, within the app-source code. The contextual analyzer on a profile call graph returns a set of detected defects, each of which contains a defect type and a sequence of evidential method signatures, ordered by time-stamps. The sequence of evidential methods, depending on the defect type as outlined in Table 17, may contain (a) *when*-context, the events related to foreground activity stages, (b) *how*-context,

Algorithm 6 Contextual Analysis for Tail-Energy Hotspot

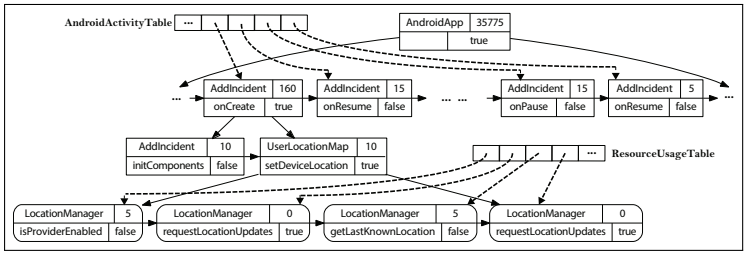
```
1: i ← 0
2: while (i < RT.size()) do
3:     if (RT[i] ∈ u_h) then                          ▷ Android API call to use h
4:         bTime ← FOREGROUNDBEGIN(AT, i)
5:         eTime ← FOREGROUNDEND(AT, i)
6:         j ← i + 1                                  ▷ to find the next resource usage index j
7:         while (RT[j].timeStamp < eTime) do
8:             if (RT[j] ∈ u_h) then
9:                 defects.info ← defects.info ∪
10:                    PROFILEGRAPHBROWSE(RT[i],RT[j])
11:                i ← j
12:            end if
13:            j ← j + 1
14:        end while
15:        if (defect is not empty) then
16:            defect.type ← "tail-energy"            ▷ new a defect
17:            ds ← ds ∪ {defect}                     ▷ add a new defect
18:        end if
19:        i ← j
20:    else
21:        i ← i + 1
22:    end if
23: end while
```

Android API calls where a hardware component is acquired, used, or released, or (c) other Android API calls not associated with the involving hardware component, but with noticeable computational time, e.g., greater than 5 milliseconds. Each evidential method α has detailed signatures which include α's prototype and its Class information, its caller's prototype and Class information, and the details of α's subsequent calls in its caller method. The subsequent calls are used to resolve the ambiguity in case that there are multiple occurrences of α in its caller method. We assume that the app-developer has access to the app source-code while debugging. To localize a defect, our framework simply highlights the defect type and displays the defect scenario by illustrating the sequence of evidential methods, and further allows users to select each of evidential method to reach the source code where it was invoked for understanding the reported defect.

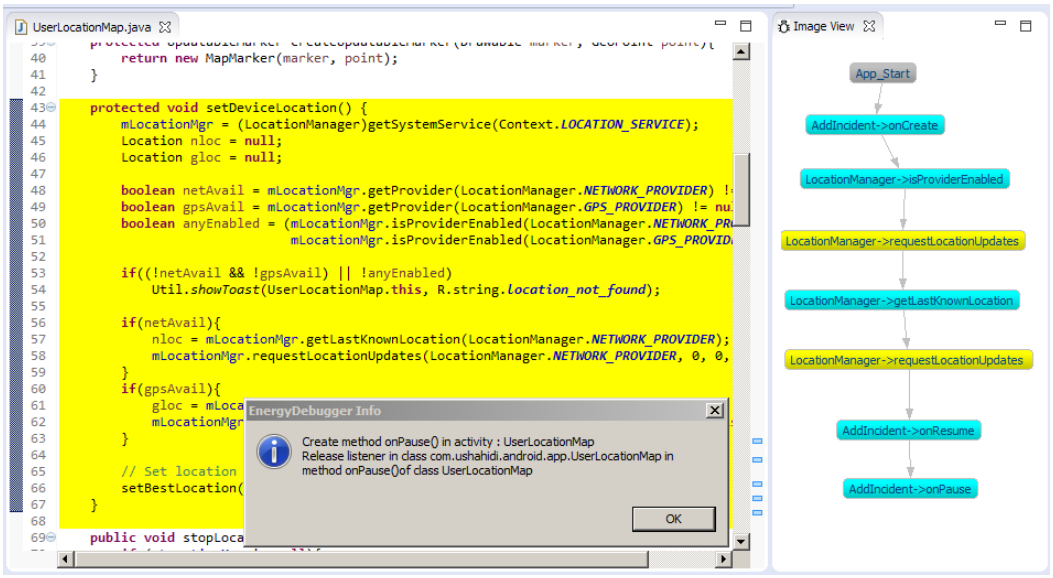
Our framework also gives concrete patch suggestions, as outlined in Table 17, for each reported defect. For example, on the defect of resource-leaks where the acquisition Android API call, a_i, misses a corresponding release statement, it is suggested that a release Android API call may be added in $onPause$. There may be multiple alternative locations (in the app-source code), where the patch may be added. Specifically, in the case of energy bugs, where the acquired services/resource/power management utilities can be released in several locations in the program. Our framework takes a conservative approach in suggesting patch-location for energy bugs, suggesting the $onPause$ event handler as the patch location for the activity that acquires the service/resource/utility, since for each activity that comes to the foreground, the $onPause$ event handler is called whenever it leaves the foreground. For another example, in the scenario of tail-energy hotspot, where usages of network components u_n are scattered with noticeable irrelevant methods X_c in-between, it is suggested that all u_n's with a same network component should be grouped together, if possible, and followed by the irrelevant methods X_c's. Similarly, for the loop-energy hotspot where tail-energy hotspot occurs within a loop structure, it is suggested that the irrelevant methods X_c's should be handled in a separated subsequent loop, if possible, for energy saving.



(a) Some GUI Elements



(b) Contextual Profile Sub-graph



(c) Snapshot showing defect localization, visualization and patch suggestion

Figure 63: Debugging Ushaihdi Android

8.3 TOOL WALK-THROUGH

In this section, we shall demonstrate our developer's debugging tool for defect detection and patch suggestion. Our framework has been embedded into an Eclipse Plugin (*EnergyDebugger*) that facilitates easy access for developers. As shown in Figure 64, the debugging process begins when the developer selects an open project in *EnergyDebugger* perspective. At this point, the selected project is parsed to obtain project metadata such as class description, method signatures, method-to-line number mappings, loop-to-line number mappings, etc. The framework also keeps track of `Activities`, `Services` and `Listeners` within the app. The code for metadata collection is implemented using JDT core (`org.eclipse.jdt.core`) libraries [216]. The debugging process then allows the developer to load a log file which is pre-generated by user, using our instrumentation and logging tool. Following automatic contextual analysis and defect localization, the debugging tool reports all found defects and their associated patch suggestions; each defect can be selected by developers for further visualization and understanding. We present a walk-through of our debugging tool for two Android apps, namely *Ushaihdi Android* [217] and *Shortyz* [218].

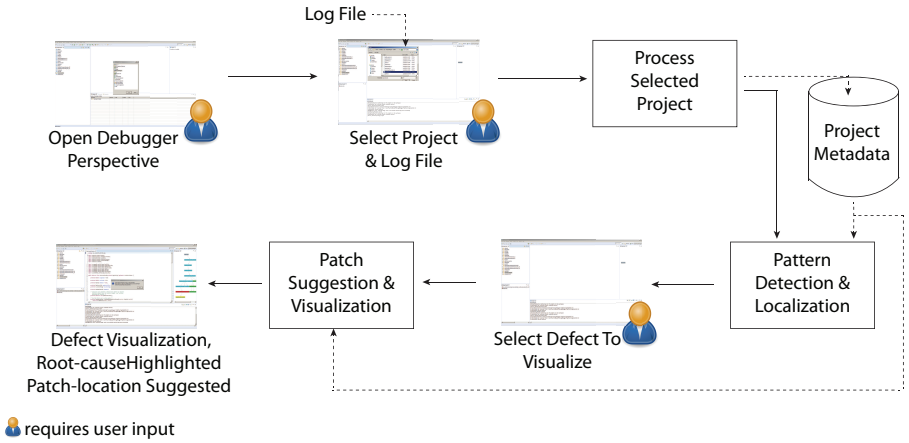


Figure 64: Working with the debugging tool in Eclipse

Ushaihdi Android is an open source app for crowd-sourcing crisis information over the internet. In our experiments we use version $v2.2$ of the app, which had a known vacuous background services defect (Issue 11). Figure 63(a) presents *some* of its GUI elements. The test case obtained from the online issue report to generate the faulty behaviour is `App Start` $\rightarrow$ `OK` $\rightarrow$ `Back` $\rightarrow$ `Add Report` $\rightarrow$ `Home`. Figure 63(c) shows a partial screenshot of our tool for debugging *Ushaihdi Android*. The screen shows the scenario when a defect of vacuous background services is found and visualized. The editor view on the left displays the source code `UserLocationMap.java`, where the `setDeviceLocation` method is highlighted. The image view on the right visualizes the defect scenario, including the *when*-context and the *how*-context, in a sequential order, and two service acquisition events are highlighted in yellow. The user may click any of those nodes in the image to jump to their respective invoked source code in the editor. The pop-up message, together with all the highlights, gives users a clear clue for patching. For this app, the suggested patch involves stopping the service in the `onPause` method of class `UserLocationMap.java`. Note that the defect scenario is derived from its corresponding contextual profile subgraph, as shown in Figure 63(b), via contextual analysis. Each evidential method node in the visualization contains a detailed method signature, which can easily redirect each method node to its invoked source location. In addition, our debugging tool provides different hierarchical views (e.g., the contextual subgraph) for users to understand defects; for the sake of concise presentation, we show a simple image view in Figure 63(c).

Our framework also performs an additional step of processing which could not be highlighted in the previous example. This step checks if the detected defect patterns falls within a loop (*i.e.* the scenario of loop-energy hotspot) or within a region of code that is launched on reboot (*i.e.* the scenario of immortality bug). We describe this step by using the *Shortyz* app. *Shortyz* is a crossword puzzle application that downloads, processes and displays free crossword puzzles from a variety of internet locations. After obtaining the log-files our framework generates the profile call graph (see Figure 62). To achieve its functionality, this app launches puzzle download procedure repeatedly (method `download` in class `Downloaders.java`). These procedures fetch puzzle data over the network (see invocations of APIs, *URL* and *HttpGet*, in Figure 62). The interesting insight which our framework provides is that a processing step (method `processDownloadedPuzzle` in class `Downloaders.java`) is triggered in between two download procedures, leading to occurrence of tail-energy hotspot in the app. Additionally, the download-processing iterations happen multiple times in a loop, as a result this app has an instance of loop-energy hotspot.

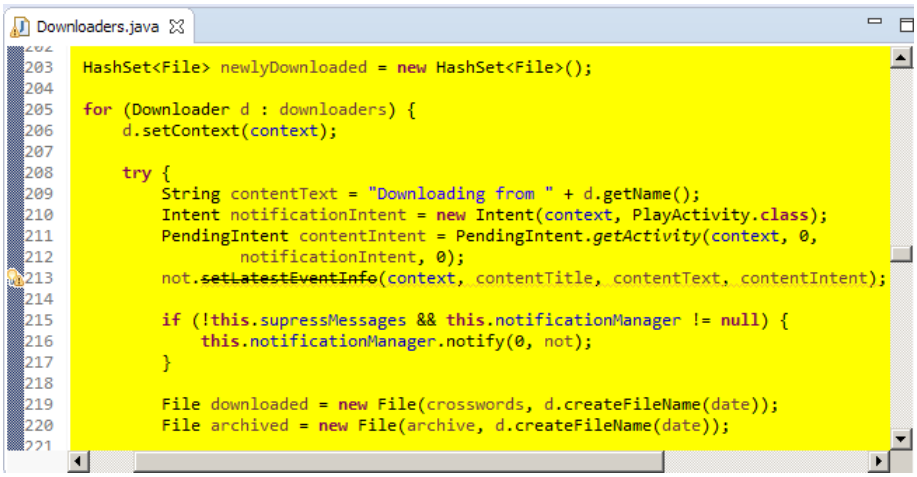


Figure 65: Defect localization for the Shortyz app

Our framework precisely locates the defect location (see Figure 65, where line 205 indicates the loop) and provides all these information (through highlighting, graphs and reports) to assist the developer in fixing the defect.

8.4 EVALUATION

We evaluated our framework to address the following research questions: (*i*) Does the log-based *field failure* localization approach, that has been used in our framework, works for real-life apps? Is this approach scalable? (*ii*) How useful are the results of defect-localization approach and the patch-suggestion approach? (*iii*) Is it possible to reduce the energy-consumption of the affected apps by applying the patches suggested by our framework?

8.4.1 Experimental Setup

We used an off-the-shelf Samsung S4 smartphone to generate log-files for our experiments. The specific device that was used in our experiments was equipped with a Quad-core 1.6 GHz CPU, 2 GB RAM and was running Android Kitkat OS (4.4.2). The instrumentation and logging utility, debugging framework, as well as a power measurement utility were all run on a PC with an Intel Core i7 processor, 8 GB RAM and Windows 7 OS. We used the setup shown in Figure 57 to measure the energy consumption of an app on the mobile device. In particular, we used a Monsoon Power Monitor [223] to supply the mobile device with a steady voltage of 4.2 Volts and to measure its power consumption. During interactions with the mobile device (when measuring energy-consumption) we have followed a few timing protocols to maintain consistency across all measurements. For instance, the interval between two successive events (touches/taps/clicks) in the test case was 15 seconds and an idle time of 45 seconds was observed just after the app had stopped execution. Also for all experiments the screen time-out duration of the device was set to 15 seconds.

8.4.2 Subject Programs

We gathered two sets of open-source apps to be used in our experiments. We shall henceforth refer to these sets as *patched-apps* and *unpatched-apps*. *Patched-apps* consist of apps for

Table 18: Open-source, Android apps that were used in the evaluation of our framework

	App Name version/code	App Description	LoC/Apk Size (KB)	Event Handler Classes	Android API Calls	Energy-intensive Resources Used*
Patched Apps	DroidAR 1.0/1 [146]	An augmented-reality app for Android.	18177/398	6	224	n, l, c, s, d
	Osmdroid 3.0.1/2[148]	Provides replacement for Android's MapView	8107/276	10	544	n, l, s, d
	Osmdroid 1.0.0/1[212]	Provides replacement for Android's MapView	5308/225	9	413	n, l, d
	Recycle-locator 1.0/1 [149]	Area-specific restroom, mailbox finding app.	717/116	3	61	n, l
	SP Transport 1.08/9 [209]	Android app that assists in bus-travel	1437/142	3	23	n, l
	SP Transport 1.17/18[150]	Android app that assists in bus-travel	1766/161	3	56	n, l
	Ushaidi v2.2/13[151]	App for Collection, visualization for crisis data	10621/713	22	276	n, l, p, c, d
Unpatched Apps	Aripuca 1.3.4/24[108]	Recording tracks and saves waypoints.	8093/660	14	730	n, l, s, d
	Benchmark 1.1.5/9[147]	Comprehensive benchmark suite for Android devices	9739/1020	23	71	n, p, d
	Shortyz 3.1.0/30100[218]	App to downloads and displays crossword puzzles	5638/175	12	568	n, d
	iTLogger 1.0.0 / 2[219]	An app for measuring road quality using on-board sensors	4014/553	7	205	l, s, b, c, p, d
	Omnidroid 0.2.1/6[123]	Automated event/action manager for Android	12425/258	28	393	n, l, c, d
	MobiPerf 2.5/1050[220]	App for doing mobile network measurements.	8009/401	12	340	n, l, p
	Sensorium 1.1.8/11[163]	Collects sensor data such as 3G, GPS, battery charge.	4001/1248	6	7840	n, l, b, s, d
	StrobeLight 1.2/3[221]	Strobe light apps using camera-flash	210/22	1	21	c
	Userhash 1.1/2[222]	View location of friends and family.	837/171	7	405	n, l, p
	Zmanim 3.3.84.296/84[152]	List of halachic/halakhic times.	72977/842	4	1102	n, l

* Network(n), Location(l), Camera(c), Sensors(s), Storage(d), Power Management(p), Bluetooth(b)

which there exist known energy-related issues reported by the user (along with test-scenarios) and patches that are applied by the developers to resolve the energy-related issue. *Unpatched-apps* on the other hand consist of apps for which energy-related issues were known (from previous works such as [159]) but there were no developer-provided patches to fix the issue. Table 18 provides some key information for both sets of apps. These apps are diverse in size and use of energy-intensive resources. The number of event-handler classes (*i.e.* related to activities, services and receivers, details in Table 16) varied from 1 to 28, whereas the number of Android API calls related to energy-intensive resources varied from 21 to $7,840$. The energy-intensive resources used by each app are shown in the last column of Table 18. The line-of-code (LoC) of these apps varied between 201 to $72,977$ (average LoC of $10,122$). The links for source-code of subject apps are provided in [224].

8.4.3 Efficacy of Defect-detection

We conducted the first set of experiments with the *patched-apps*. These apps were specifically useful in the initial stages of this work because they provided us with useful insights into how users observed and reported energy-related issues, as well as, how developers analysed and fixed these issues. A list of these issues can be found in column 2 of Table 19. We observed that when these issues were brought to developer's notice they were classified with *medium*

or *high* priorities. Also the developers usually fixed these defects within a fortnight. However, in some cases issues remained open for a considerably long period of time, mostly due to improper understanding of the defect. Consider the Issue 53 [148] in app Osmdroid, which stayed open for 240 days during which time 559 versions were committed to the repository. We present the timeline of the conversation related to Issue 53 in following.

May 25, 2010	Issue opened. Test case provided.
Oct 4, 2010	Priority increased from *low* to *medium*
Oct 8, 2010	User thinks defect is fixed
Jan 11, 2011	Project member suggests closing the issue
Jan 12, 2011	Project member claims the issue is not fixed
Jan 20, 2011	Project member fixes Issue. Provides commit number were issue has been fixed

Unlike the confusion which ensues as a result of ad-hoc communication, our framework provides definitive answers as to whether there exists a defect for a given test case/scenario. When we tested the log-files generated from the apps (including for Osmdroid Issue 53) our framework was able to pinpoint exact locations of defects in the source-code (manually cross-checked by going through user-comments, code-changes, etc). Also the defect visualization provided by our framework further assists the developer in understanding the cause and effect of the observed defect. Once confident about the efficacy of our framework (in detecting defects), we conducted another set of experiments with the *unpatched-apps* for which our framework was able to pinpoint locations of the defects in the source-code as well. Table 21 provides a list of defect-locations for this set of apps.

Table 19: Summary of defect localization and patch location suggestion for patched-apps

App Name version /code	Issue No, Defect Description, Assigned Priority	Issue Open Days	Commits While Open	Observed Changes Class (Method)	Proposed Changes Class (Method)	Energy Saved (%)
DroidAR 1.0 / 1	Issue 27, Vacuous background services Priority-Medium	3	2	Commit : 2*e*315*c*080974*a*56751*e* EventManager.java(resumeEventListeners) **Setup.java**(onDestroy,onStop,onRestart killCompleteApplicationProcess,**onPause**) ArActivity.java(onResume), GeoUtils.java(disableGPS)	Setup.java (onPause)	29
Osmdroid 3.0.1 / 2	Issue 53, Vacuous background services Priority-Medium	240	559	Commit : *r*751 **SampleMapActivity.java**(onResume, **onPause**)	SampleMapActivity.java (onPause)	11
Osmdroid 1.0.0 / 1	Issue 76, Expensive background service Priority-Medium	11	4	Commit : *fd*2*b*17227*ab*183*a*5*a*16 **MyLocationOverlay.java** (getLocationUpdateMinTime, getLocationUpdateMinDistance, **enableMyLocation**, setLocationUpdateMinDistance)	MyLocationOverlay.java (enableMyLocation)	5
Recycle Locator 1.0 / 1	Issue 33, Vacuous background services Priority-Medium	0	0	Commit : *f*19*b*7*fc*5*a*7*a*0 **Map.java**(**onPause**, createMap,locateUser)	Map.java (onPause)	23
SP Transport 1.08 / 9	Issue 38, Sub-optimal resource binding Priority-Medium	2	9	Commit : 698*a*27*e*83900*e*42*a*6*dd* LocationView.java (disableLocationUpdates,onResume, enableLocationUpdates) **HtmlResult.java** (**showResult**)	HtmlResult.java (showResult)	17
SP Transport 1.17 / 18	Issue 76, Vacuous background services Priority-High	0	0	Commit : *d*2*dfa*786*da*728*ae*6975 **LocationView.java** (**onPause**, onCreateDialog)	LocationView.java (onPause)	12
Ushaidi v2.2 / 13	Issue 11, Vacuous background services Priority-n/a	0	0	Commit : 561*d*6*fb*10*a*5*fc*600*ab*6 **UserLocationMap.java** (**onPause**, onDestroy)	UserLocationMap.java (onPause)	10

8.4.4 Scalability of Defect-detection

The scalability of our technique is mostly dictated by the sizes of the log-files that are used for profile generation. In general, programs with more event-handlers and Android API usages may generate bigger log-files. The length of test-input sequence may also affect the size of log-files. It is worthwhile to know that LoC may not be a good indicator of log-file size, as it may not correspond to more event-handlers or API usages. For example, *Shortyz* has only 5,538 LoC but generated 50,049 lines of log messages whereas *Benchmark* has 9,739 LoC but generated only 356 lines of log messages (*cf.* Tables 20 and 21). In general, the analysis was relatively fast, with the longest analysis time being 3.1 seconds for 112,897 lines of log messages for *DroidAR*.

Table 20: Line of log messages and analysis time for all apps

	App Name (version/code)	Line of Log Messages	Analysis Time (seconds)
Patched Apps	DroidAR (1.0 / 1)	112,897	3.1
	Osmdroid (3.0.1 / 2)	3,504	0.1
	Osmdroid (1.0.0 / 1)	1,526	0.1
	Recycle-locator (1.0 / 1)	396	0.1
	SP Transport (1.08 / 9)	543	0.1
	SP Transport (1.17 / 18)	14,281	0.2
	Ushaidi(v2.2 / 13)	876	0.1
Patched Apps	Aripuca (1.3.4 / 24)	3,838	0.5
	Benchmark (1.1.5/ 9)	356	0.1
	Shortyz (3.1.0 / 30100)	50,049	0.6
	iTLogger (1.0.0 / 2)	2,967	0.1
	Omnidroid (0.2.1 / 6)	2,284	0.2
	MobiPerf (2.5 / 1050)	3,210	0.2
	Sensorium (1.1.8/ 11)	11,347	0.5
	StrobeLight (1.2 / 3)	87	0.1
	Userhash (1.1 / 2)	226	0.1
	Zmanim (3.3.84.296 / 84)	50,892	0.8

8.4.5 Effectiveness of the Patch-suggestion

The final set of experiments (with a power-measurement setup as described in Section 8.4.1) were conducted to evaluate the effectiveness of the patch-suggestion. For the *patched-apps* this was relatively straightforward as we could compare the patches suggested by our framework to the patches applied by the developer for each defect. However, for *unpatched-apps*, since there was no such information available, we compared energy consumption before and after patches were applied, to evaluate the effectiveness of the patches. For the experiments with *patched-apps*, we observed a significant correlation between the changes suggested by our framework and changes added to the source-code commits where the defect was resolved (*cf* Table 19, columns 5 and 6). Power measurement experiments with the *unpatched-apps* also returned positive results, with energy-savings varying from 6% to 29%.

Table 21: Summary of results for unpatched apps

App Name version/code	Defect Description	Defect Location	Energy Saved (%)
Aripuca 1.3.4/24	Vacuous services	AppService.java(startLocationUpdates, startSensorUpdates)	15 15
Benchmark 1.1.5/9	Wakelock Bug	Benchmark(onCreate)	29
Shortyz 3.1.0/30100	Loop-energy Hotspot	Downloaders.java(processDownloadedPuzzle, download) + 3 more	11 11
iTLogger 1.0.0/2	Vacuous Services	iTloggerActivity(onCreate)	9
Omnidroid 0.2.1/6	Immortality Bug	LocationMonitor(init)	14
MobiPerf 2.5/1050	Wakelock Bug	PoneUtils(acquireWakelock)	12
Sensorium 1.1.8/11	Vacuous Services	GPSLocationSensor(_enable), NetworkLocationSensor(_enable)	21
StrobeLight 1.2/3	Sub-optimal Binding	StrobeRunner(run)	6
Userhash 1.1/2	Immortality Bug	ViewActivity(onClick)	15
Zmanim 3.3.84.296/84	Resource Leak	LogUtils(startSession)	21

8.5 COMPARISON WITH EXISTING WORKS

Due to increasing popularity of smartphones and their limited battery power, energy consumption models of mobile apps have been extensively studied [203, 225, 179, 180, 181, 182, 226, 192] for monitoring and optimizing their energy usage. These models, typically based on monitoring device components' measurable parameters and computing the approximate battery drain caused by the component over time, provide an energy-aware quality of service support for users.

Increasing attention has been drawn to the research connecting energy consumption models with various user or system activities in mobile applications. For examples, real user activities [183, 184] were collected and studied to guide power consumptions for mobile architectures. An accurate power model was constructed based on both the power management and activity states of those power-intensive hardware components [225]. Eprof [185, 226] is a fine-grained energy profiler for smartphone apps, mapping the power draw and energy consumption to program entities via a Android API driven finite state machine [226]. PowerScope [186] is an alternative tool for profiling the energy Usage of Mobile Applications. However, those energy profiling tools still focus on answering the ultimate question, "*Where is the energy spent inside my app?*", to provide an energy-aware quality of service support.

Several automated tools [227, 189, 228] have been recently developed for detecting energy anomalies and diagnosing energy inefficiency for mobile applications. The ADEL tool [227] detects and isolates energy leaks resulting from unnecessary network communication via data flow analysis. The Carat tool [189] detects and diagnose energy anomalies by using a collaborative battery consumption and utilization measurements aggregated from multiple clients. The GreenDroid tool [228] monitors sensor and wake lock operations to detect two common causes of energy problems: missing deactivation of sensors or wake locks, and cost-ineffective us of sensory data, and generate detailed reports to assist developers in validating detected energy problems.

Debugging functionality-related defects, e.g., fault localization [199, 229] and patch generation [198, 200, 201], is difficult and time-consuming. Debugging energy-inefficiency defects is even more challenging because the defects may depend on the running contextual sensitive scenario, including the state of hardware device, a specific sequence of user interactions, and program call dependencies. For locating energy-inefficiency issues, developers often seek assistances from users through online coding repositories since no debugging tool is currently available. Our framework provides both an effective communication channel from users to developers and an automated energy-inefficiency debugging tool.

8.6 CHAPTER SUMMARY

We present a practical framework for localizing energy-efficiency related *field failures* in mobile apps. Our framework provides a simple yet effective communication protocol to be used between the users and developers. Not only does our developer-side debugging tool detect and localize the presence of energy defects in the app source code, but also suggests potential patch location for the reported defects. Evaluation with real-life apps have shown that our tool can localize defects effectively and efficiently, even for apps with thousands of lines-of-code. Additionally, the energy savings generated as a result of the patched defects are significant (observed energy saving between 5% to 29%).

9 REFLECTIONS

In this chapter we shall briefly discuss the key contributions of this thesis. We shall also discuss a few potential directions for future work that can be done to address the challenges faced by contemporary app-development companies. The information for the future work was gathered through interactions with programmers and managers from commercial mobile app-development companies in Singapore.

DISCUSSION

Systematic non-functional testing was a relatively less explored topic hitherto. However, as we have discussed in previous chapters, testing of non-functional properties is equally important as testing of functional properties, especially in the context of real-time systems which may have to operate under a number of non-functional constraints. This work is an effort to address this situation. Our work does not try to re-invent automated testing in context of non-functional properties, instead it uses novel abstractions (such as the automated instrumentation of assertions in Chapters 3 and 6) to capture the peculiarities commonly associated with non-functional properties such as performance and energy-consumption. These abstractions helps us to develop sound and precise frameworks based on well-accepted systematic testing technique such as DART[5]. We have also discussed a number of practical applications of the non-functional testing frameworks that were described in the previous chapters. The most obvious application being the fact that programmers/testers can now systematically generate non-functional property aware test-cases (for instance, performance aware test-case generation in Chapter 3 and energy-consumption aware test-case generation in Chapters 5 and 6). Before our techniques were developed, the only way a programmer/tester could have done non-functionality aware testing was through profiling. However, as we have discussed in Chapters 1, 2 and 4 that unlike test-generation techniques, profiling techniques need to be provided with test-cases for them to generate the profile which can then be analysed to study the non-functional behaviour of the program. This brings us to another important observation; it is difficult to generate non-functional behaviour stressing test-cases manually. So if a programmer/tester were trying to uncover suboptimal non-functional behaviour using just profiling techniques, it would be a challenging and time-consuming task in the best of scenarios and an impractical task in the worst of scenarios. This is not to say that profiling techniques are not useful, on the contrary they can hugely benefit from a systematic test-generation technique such as ours and further assist the programmer in understanding the behaviour of a program. We have also explored some of the less-obvious applications of non-functionality aware test-case generation in design space exploration and cache locking in context of performance optimization for real-time embedded systems in Chapter 3 and in energy-efficient repair generation for mobile-apps in Chapter 6. Specifically the works on mobile-app testing (Chapters 4 - 6) have provided us with insights that have enabled us to extend the support for energy-aware programming to development (energy-aware re-factoring framework presented in Chapter 7) and debugging (energy-aware re-factoring framework presented in Chapter 8) stages of the app development. Equipped with these new tools and techniques for systematic non-functional testing we next seek to solve some of the testing related problems faced by app-developers in commercial app-development companies.

CHALLENGES IN MOBILE-APP TESTING IN COMMERCIAL SETTINGS AND FUTURE WORK

As of year 2015 the market size for mobile apps has reached a staggering 40 billion dollars and due to the popularity of smartphone it is expected to keep growing to 76 billion dollars by the year 2017 [230]. The research community has also been very active in the last few years in generating tools and techniques that can facilitate in processes related to development, testing, rating and recommendation of mobile apps. Research works presented in Chapter 4 can provide an idea about some of these works. Interestingly, the market for mobile app test automation has been forecasted to grow from 200 million dollars in 2015 to approximately 800 million dollars by the year 2017 [231]. However, what we are interested to find out is (i) how much of the research advancements in the academia have been successfully adopted in the industry? (ii) what kind of testing automation services are available in the commercial setting? and (iii) what testing-related problems are currently faced by app-developer in spite of the available tools and techniques? In the following paragraphs we shall provide a short summary of some of the interviews we have had with professionals from the app-development industry in our pursuit to answer these questions.

Testing is often one of the ways by which developers/clients ascertain/judge the quality of their apps. So to know if automated testing (or testing of any sort) had any importance in the industry we had to first find out whether app-developers cared about the quality of their apps. We found that app-developers do care about quality (and testing) but to varying extent; the extent being decided based on the specific requirements imposed by the clients (of the app-development companies). In general, small-medium scale enterprises (SMEs) are more interested in the functional aspects of their app whereas the enterprise clients are more stringent about functionality as well as quality, specially when the app functionality involves using credit card information[1]. We use the term *quality* rather vaguely in this chapter, however as it turns out the actual significance of the term quality is also vaguely defined in the commercial setting and may depend on the context. For instance, higher quality may indicate higher performance when it comes to gaming apps whereas higher energy-efficiency may indicate higher quality for apps that run for long durations of time. However, certain aspects of quality, such as app does not crashes for common UI inputs, was often agreed to be a part of the bare minimum definition of quality. So for the rest of this chapter we shall restrict ourself to this bare minimum (and still vague) definition of quality as a measure of how well an app has been validated against crashes. Interestingly, even though the body of research work on the topic of automated mobile app testing has been growing by leaps and bounds, not much of the recent advancements have permeated to the commercial app testing setting. Much of the testing work in the commercial setting is still done using non-mobile-specific frameworks such as JUnit[233] and JMeter[234]. It is worthwhile to point out that these observation may not represent the global, commercial app development community, but a characteristic of the local app development community. We have also observed that one of the key challenge faced by developers while using manual test-suite design techniques was in coming up with appropriate test-suite. As is the case with most real-life software, manually crafting adequate test-suite requires intimate knowledge of the software system as well as time. Both of which may be expensive resources in the commercial setting. These challenges to adequate app testing are increased by the fact that in a commercial setting often (mostly for SMEs) the UI components may be changed, re-organized or re-designed a number of times during the project. This may often require the test-suite to updated or re-created, further increasing the

1 Payment Card Industry Data Security Standard (PCI DSS) must be followed when cards such as Visa, MasterCard, American Express are handled by the app [232]

costs associated with testing. Added to these challenges is the fact there are many smartphone devices for both iOS and Android platforms, each of which may be running a different version of OS and have different hardware (such as screen size). All these factors combined lead to a situation where adequate app-testing is often uneconomical and therefore app-development companies often have to rely on skilfully crafted client-company agreements to deliver their products (apps) on time and within budget.

We also looked at the different products and services that are currently being provided for mobile app testing. Table 22 provides a list of the leading mobile-app testing companies along with the products/services that they provide and an estimated price for these products/services. In particular, they provide one or more of the following services:

- Help in test-generation: Fully-automated testing is probably the most desirable feature which the app-development companies would be looking for in commercial testing based products. However, none of the companies that we have looked at offered fully-automated testing based products. Instead, they provide different mechanisms to assist the app-developers to manually craft test-cases. These mechanisms can be put in following two categories:
  - HTG-A: Developer provides test scenarios through an easy-to-use interface. These manually generated test scenarios are then converted into test-cases
  - HTG-B: Developer interacts with the app during which a system records the execution trace to generate test-cases
- Cloud-based testing framework: testing products/services are provided as cloud-based platforms such that app developer can access it from anywhere
- Testing on multiple devices: developer may want to test their app on multiple devices. However, buying multiple devices (smartphone) can be uneconomical and testing the app on these multiple device one-by-one is time consuming. Therefore, commercial app testing companies often provides the app-developers access to many devices on which to test their app on. Note that the testing is still done using the test-cases provided by the app-developer using approach HTG-A/B.
- Crowd sourcing: These services provide a platform through which app developers and app users can interact. New apps are uploaded through the service and app users provide detailed reviews on the look and feel of the app. This form of testing is more suitable for the later stages of the app development (say during beta testing).

Table 22: Products offered by mobile-app testing companies (data collected on 4th September 15)

Company Name	**Testing on Multiple Devices**	**Cloud Testing**	**Help in Test Generation**	**Price Range Basic - Enterprise**
Keynote	Yes	Yes	HTG-A	180 $ - 750 $ Month
Ranorex	Yes	No	HTG-A/B	700 $ - 3,500 $ License
TestDroid	Yes	Yes	HTG-B	49 $ - 3999 $ Month
Experitest	developer buys device	setup on user facility	HTG-A	1000 $ Year
PerfectoMobile	Yes	Yes	HTG-B	99$ - $299 Month
UserTesting	Crowd sourcing	n/a	n/a	49$ - 99$ User Review
Xamarin	Yes	Yes	HTG-B	1,000$ - 12,000$ Month

Interestingly, the approaches provided by the academia as well as by the industry both claim to offer solution for test automation. So naturally the question arises as to which of the

automation approaches is better? Probably, an even more important question is which of these two approaches better solves the challenges faced by app-developers in the commercial setting? At this point it is worthwhile to point out that the description of *test automation* widely varies from the academia (as in academic research papers) to that used in commercial testing based products/services in the industry. In general, in the academic setting *test automation* usually implies that given a program the testing-framework can explore the program and generate failure-revealing test-cases, automatically (with little or no user effort). Hence, the academic notion of *test automation* often involves creating abstract representation of the program (to be tested) and ways to systematically explore this abstract representation to generate failure-revealing test-cases, under the supervision of a failure-predicting oracle. On other hand, the industrial (as in companies that provide app-testing services) notion of *test automation* usually involves ornate mechanisms (see HTG-A/B described in previous paragraph) through which the app-developer can specify the test-cases manually. Such a notion of *test automation* requires the app-developer to be skilled in the art of testing and also spend time in creating the test-cases. Even though the cloud-based testing services as well as access to wide-range of testing devices are desirable features provided by existing app-testing companies, without a more comprehensive *test automation* strategy existing commercial mobile app testing services do not solve the challenges faced by the app-developer adequately. While the academic notion of *test automation* can better address the challenges faced by commercial app-development community, they are often less-accessible (no cloud-based framework) and are often complicated to use. It would be interesting to see a solution where the research advances in *test automation* are integrated with a cloud-based testing framework such that app-developers can test their apps on many devices, with least amount of configuration. Such a framework would be very useful to app developers as it will reduce the costs associated with testing while generating comprehensive test results. We wish to investigate development of such a framework in our future work.

BIBLIOGRAPHY

[1] Abhijeet Banerjee, Sudipta Chattopadhyay, and Abhik Roychoudhury. Static analysis driven cache performance testing. In *Proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS 2013, Vancouver, BC, Canada, December 3-6, 2013*, pages 319–329, 2013.

[2] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 588–598, New York, NY, USA, 2014. ACM.

[3] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[5] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[6] Yun Liang and Tulika Mitra. Instruction cache locking using temporal reuse profile. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 344–349, New York, NY, USA, 2010. ACM.

[7] Jean-FranÃğois Deverge and Isabelle Puaut. Safe measurement-based wcet estimation, 2005.

[8] Guillem Bernat, Antoine Colin, and Stefan M. Petters. Wcet analysis of probabilistic hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, RTSS '02, pages 279–, Washington, DC, USA, 2002. IEEE Computer Society.

[9] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 67–76, New York, NY, USA, 2012. ACM.

[10] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 89–98, New York, NY, USA, 2012. ACM.

[11] Christopher Healy, Mikael SjÃűdin, Viresh Rustagi, David Whalley, and Robert Van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Journal of Real-Time Systems*, 18:121–148, 2000.

[12] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 136–146, Washington, DC, USA, 2009. IEEE Computer Society.

[13] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, pages 358–363, New York, NY, USA, 2006. ACM.

[14] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference*, DAC '95, pages 456–461, New York, NY, USA, 1995. ACM.

[15] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise wcet prediction by separated cache andpath analyses. *Real-Time Syst.*, 18(2/3):157–179, May 2000.

[16] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, RTSS '96, pages 254–, Washington, DC, USA, 1996. IEEE Computer Society.

[17] Damien Hardy and Isabelle Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of the 2008 Real-Time Systems Symposium*, RTSS '08, pages 456–466, Washington, DC, USA, 2008. IEEE Computer Society.

[18] Sudipta Chattopadhyay and Abhik Roychoudhury. Scalable and precise refinement of cache timing analysis via model checking. In *Proceedings of the 2011 IEEE 32Nd Real-Time Systems Symposium*, RTSS '11, pages 193–203, Washington, DC, USA, 2011. IEEE Computer Society.

[19] Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '98, pages 16–30, London, UK, UK, 1998. Springer-Verlag.

[20] Rathijit Sen and Y. N. Srikant. Wcet estimation for executables in the presence of data caches. In *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '07, pages 203–212, New York, NY, USA, 2007. ACM.

[21] S. Chattopadhyay and A. Roychoudhury. Unified cache modeling for wcet analysis and layout optimizations. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 47–56, Dec 2009.

[22] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Wcet analysis of multi-level set-associative data caches. In *Workshop on WCET*, 2009.

[23] Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. Scope-aware data cache analysis for wcet estimation. In *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '11, pages 203–212, Washington, DC, USA, 2011. IEEE Computer Society.

[24] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. An accurate worst case timing analysis for risc processors. *IEEE Trans. Softw. Eng.*, 21, 1995.

[25] S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A worst case timing analysis technique for multiple-issue machines. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, 1998.

[26] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, RTSS '95, pages 288–297, 1995.

[27] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, RTSS '95, pages 298–, Washington, DC, USA, 1995. IEEE Computer Society.

[28] Jörn Schneider and Christian Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '99, pages 35–44, New York, NY, USA, 1999. ACM.

[29] Marc Langenbach, Stephan Thesing, and Reinhold Heckmann. Pipeline modeling for timing analysis. In *Proceedings of the 9th International Symposium on Static Analysis*, SAS '02, pages 294–309, London, UK, UK, 2002. Springer-Verlag.

[30] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for wcet analysis. *Journal of Real-Time Systems*, 34:2006, 2005.

[31] J. Engblom. Analysis of the execution time unpredictability caused by dynamic branch prediction. In *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE*, pages 152–159, May 2003.

[32] Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor withbranch prediction. *Real-Time Syst.*, 18(2/3):249–274, May 2000.

[33] Xianfeng Li, Tulika Mitra, and Abhik Roychoudhury. Modeling control speculation for timing analysis. *Real-Time Systems*, 29, 2005.

[34] Daniel Grund, Jan Reineke, and Gernot Gebhard. Branch target buffers: Wcet analysis framework and timing predictability. *J. Syst. Archit.*, 57, 2011.

[35] Y-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Trans. Des. Autom. Electron. Syst.*, 4(3), 1999.

[36] Francois Bodin and Isabelle Puaut. A wcet-oriented static branch prediction scheme for real time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, ECRTS '05, 2005.

[37] Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48:53–70, 1999.

[38] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '01, 2001.

[39] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *RTSS*, 2000.

[40] Reinhard Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In *VMCAI*, 2004.

[41] MINISAT satisfiability solver. http://minisat.se/.

[42] J. Gustafsson et al. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *RTSS*, 2006.

[43] V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *DAC*, 2008.

[44] Y. Li et al. Timing analysis of concurrent programs running on shared cache multi-cores. In *RTSS*, 2009.

[45] D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS*, 2008.

[46] X. Li et al. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 2007. http://www.comp.nus.edu.sg/~rpembed/chronos.

[47] WCET benchmarks. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[48] F. Nemer, H. Cassé, and P. Sainrat. Papabench: a free real-time benchmark. *WCET Workshop*, 2006.

[49] Muhammad Yasir Qadri, Dorian Matichard, and Klaus D. McDonald Maier. Jetbench: An open source real-time multiprocessor benchmark. In *Proceedings of the 23rd International Conference on Architecture of Computing Systems*, ARCS'10, pages 211–221, Berlin, Heidelberg, 2010. Springer-Verlag.

[50] Rhapsody. http://www-01.ibm.com/software/awdtools/rhapsody/.

[51] CTAS case study overview, requirements. In *SCSEM Case Study*, 2003.

[52] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. *TACAS*, 2004.

[53] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. In *in ICSE*, 2009.

[54] Sudipta Chattopadhyay, Lee Kee Chong, and Abhik Roychoudhury. Program performance spectrum. In *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '13, pages 65–76, New York, NY, USA, 2013. ACM.

[55] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[56] LLVM compiler infrastructure. http://llvm.org/.

[57] KLEE symbolic virtual machine. http://klee.llvm.org/.

[58] STP constraint solver. https://sites.google.com/site/stpfastprover/.

[59] M. Harman, Y. Jia, and Y. Zhang. Achievements, open problems and challenges for search based software testing. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, 2015.

[60] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[61] James R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 259–269, New York, NY, USA, 1999. ACM.

[62] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power analysis of embedded software: A first step towards software power minimization. In *Proceedings of the 1994 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '94, pages 384–390, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[63] J.T. Russell and M.F. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. In *Computer Design: VLSI in Computers and Processors, 1998. ICCD '98. Proceedings. International Conference on*, 1998.

[64] W. Ye, N. Vijaykrishnan, and M. J. Irwin. The design and use of simplepower: A cycle-accurate energy estimation tool. pages 340–345, 2000.

[65] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, 2000.

[66] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 1997.

[67] Johann Laurent, Eric Senn, Nathalie Julien, Eric Martin, and The Pennsylvania State University CiteSeer Archives. High-level energy estimation for dsp systems. 2001.

[68] Eric Senn, Johann Laurent, Nathalie Julien, and Eric Martin. Softexplorer: estimating and optimizing the power and energy consumption of a c program for dsp applications. *EURASIP J. Appl. Signal Process.*, 2005.

[69] H. Blume, M. Schneider, and T. G. Noll. Power estimation on functional level for programmable processors. In *Advances in Radio Science*, 2004.

[70] Johann Laurent, Nathalie Julien, Eric Senn, and Eric Martin. Functional level power analysis: An efficient approach for modeling the power consumption of complex processors. In *Proceedings of the Conference on Design, Automation and Test in*

Europe - Volume 1, DATE '04, pages 10666–, Washington, DC, USA, 2004. IEEE Computer Society.

[71] Nand Kumar, Srinivas Katkoori, Leo Rader, and Ranga Vemuri. Profile-driven behavioral synthesis for low-power vlsi systems. *IEEE Des. Test*, 1995.

[72] Feng Qian, Zhaoguang Wang, Alexandre Gerber, Zhuoqing Mao, Subhabrata Sen, and Oliver Spatscheck. Profiling resource usage for mobile applications: A cross-layer approach. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 321–334, New York, NY, USA, 2011. ACM.

[73] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 92–101, Piscataway, NJ, USA, 2013. IEEE Press.

[74] Ding Li, Shuai Hao, William G. J. Halfond, and Ramesh Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 78–89, New York, NY, USA, 2013. ACM.

[75] Ramkumar Jayaseelan, Tulika Mitra, and Xianfeng Li. Estimating the worst-case energy consumption of embedded software. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '06, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[76] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel P. Midkiff. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[77] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1013–1024, New York, NY, USA, 2014. ACM.

[78] Emina Torlak and Satish Chandra. Effective interprocedural resource leak detection. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 535–544, New York, NY, USA, 2010. ACM.

[79] Timo Hönig, Christopher Eibel, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. Seep: Exploiting symbolic execution for energy-aware programming. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, HotPower '11, pages 4:1–4:5, New York, NY, USA, 2011. ACM.

[80] Dacong Yan, Shengqian Yang, and A. Rountev. Systematic testing for resource leaks in android applications. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, 2013.

[81] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

[82] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 59:1–59:11, New York, NY, USA, 2012. ACM.

[83] Yepang Liu, Chang Xu, S.C. Cheung, and Jian Lu. Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *Software Engineering, IEEE Transactions on*, 40(9):911–940, Sept 2014.

[84] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM.

[85] Irene Manotas, Lori Pollock, and James Clause. Seeds: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 503–514, New York, NY, USA, 2014. ACM.

[86] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. Energy types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 831–850, New York, NY, USA, 2012. ACM.

[87] Business insider: Smartphone and tablet penetration. http://www.businessinsider.com/smartphone-and-tablet-penetration-2013-10?IR=T.

[88] A. Memon, I. Banerjee, and A. Nagarajan. Gui ripping: reverse engineering of graphical user interfaces for testing. In *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*, pages 260–269, Nov 2003.

[89] Android develeoper website, wifimanager. http://developer.android.com/reference/android/net/wifi/WifiManager.WifiLock.html.

[90] Android application coding guidelines -power save. http://dl-developer.sonymobile.com/documentation/dw-300012-Android_Power_Save.pdf.

[91] M. Gottschalk, M. Josefiok, J. Jelschen, and A. Winter. Removing energy code smells with reengineering services. 2012.

[92] Android develeoper website, powermanager. http://developer.android.com/reference/android/os/PowerManager.html.

[93] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, pages 280–293, New York, NY, USA, 2009. ACM.

[94] Android develeoper website, sensormanager. http://developer.android.com/reference/android/hardware/SensorManager.html.

[95] Android-sensors. http://developer.android.com/guide/topics/sensors/sensors_overview.html.

[96] Android develeoper website, location strategies. http://developer.android.com/guide/topics/location/strategies.html.

[97] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 5:1–5:6, New York, NY, USA, 2011. ACM.

[98] JMOTIF: a time series data-mining toolkit based on SAX and TFIDF statistics. http://code.google.com/p/jmotif/.

[99] Business insider: Number of smartphones worldwide. http://www.businessinsider.com/15-billion-smartphones-in-the-world-22013-2?IR=T.

[100] Android power profiles. http://source.android.com/devices/tech/power.html.

[101] Hierarchy viewer. http://developer.android.com/tools/help/hierarchy-viewer.html.

[102] Eamonn Keogh and Jessica Lin. Hot sax: Efficiently finding the most unusual time series subsequence. In *ICDM*, pages 226–233, 2005.

[103] W. E. Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. In *Proceedings of the Section on Survey Research*, 1990.

[104] Yokogawa wt210 digital power meter. http://tmi.yokogawa.com/us/products/digital-power-analyzers/power-measurement-application-software/wtviewer-for-wt210wt230/.

[105] A.S. Tanenbaum and M. van Steen. *Distributed systems: principles and paradigms*. Pearson Prentice Hall, 2007.

[106] Android advanced geocachingtool. https://play.google.com/store/apps/details?id=com.zoffcc.applications.aagtl.

[107] Android battery dog. https://play.google.com/store/apps/details?id=net.sf.andbatdog.batterydog.

[108] Aripuca. https://f-droid.org/repository/browse/?fdid=com.aripuca.tracker.

[109] Kitchen timer. https://play.google.com/store/apps/details?id=com.leinardi.kitchentimer.

[110] Montreal transit. https://play.google.com/store/apps/details?id=org.montrealtransit.android.

[111] Npr news. https://play.google.com/store/apps/details?id=org.npr.android.news.

[112] Pedometer. https://play.google.com/store/apps/details?id=name.bagi.levente.pedometer.

[113] Simple chess clock. https://play.google.com/store/apps/details?id=com.chessclock.android.

[114] Wifi advanced config editor. https://play.google.com/store/apps/details?id=org.marcus905.wifi.ace.

[115] World clock. https://play.google.com/store/apps/details?id=com.irahul.worldclock.

[116] Sensor status. https://play.google.com/store/apps/details?id=com.tpaln.snsst.

[117] Zoom camera. https://play.google.com/store/apps/details?id=ar.com.moula.zoomcamera.

[118] Voice recorder. https://play.google.com/store/apps/details?id=si.matejpikovnik.voice.pageindicator.

[119] Virtual recorder. https://play.google.com/store/apps/details?id=ix.com.android.VirtualRecorder.

[120] Quick recorder. https://play.google.com/store/apps/details?id=com.workspace.QuickRecorder.

[121] Speedometer. https://play.google.com/store/apps/details?id=com.bjcreative.tachometer.

[122] Zmanim. https://play.google.com/store/apps/details?id=com.gindin.zmanim.android.

[123] Omnidroid. https://f-droid.org/wiki/page/edu.nyu.cs.omnidroid.app.

[124] Fox news. https://play.google.com/store/apps/details?id=com.foxnews.android.

[125] Best unit converter. https://play.google.com/store/apps/details?id=simple.a.

[126] Sensor tester. https://play.google.com/store/apps/details?id=com.dicotomica.sensortester.

[127] Eponte. https://play.google.com/store/apps/details?id=com.amoralabs.eponte&hl=en.

[128] Goodreads. https://play.google.com/store/apps/details?id=com.goodreads.

[129] Food court. https://play.google.com/store/apps/details?id=com.eksavant.fc.ui.

[130] Fire and blood. https://play.google.com/store/apps/details?id=com.zeddev.plasma2.

[131] 760 kfmb am. https://play.google.com/store/apps/details?id=com.airkast.KFMBAM.

[132] Math workout. https://play.google.com/store/apps/details?id=com.akbur.mathsworkout.

[133] Vanilla music. https://play.google.com/store/apps/details?id=ch.blinkenlights.android.vanilla.

[134] Vimeo. https://play.google.com/store/apps/details?id=com.vimeo.android.videoapp.

[135] Baby solid food. https://play.google.com/store/apps/details?id=com.bytecontrol.diversification.

[136] GSM Arena. http://www.gsmarena.com/lg_optimus_l3_e400-4461.php.

[137] EnergyPatch. https://bitbucket.org/abhijeet_code/energypatch.

[138] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[139] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. JPF-SE: A symbolic execution extension to java pathfinder. In *TACAS*, 2007.

[140] Bsd-3-clause license. http://opensource.org/licenses/BSD-3-Clause.

[141] Asm, java bytecode manipulation and analysis framework. http://asm.ow2.org/.

[142] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2000.

[143] S. Anand, C S Păsăreanu, and W. Visser. JPF-SE: A symbolic execution extension to java pathfinder. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2007.

[144] Heila van der Merwe, Brink van der Merwe, and Willem Visser. Verifying android applications using java pathfinder. *ACM SIGSOFT Software Engineering Notes*, 2012.

[145] Bablesink. https://github.com/hatstand/babblesink/.

[146] Droidar, issue 27. https://code.google.com/p/droidar/issues/detail?id=27.

[147] 0xbenchmark. https://f-droid.org/wiki/page/org.zeroxlab.zeroxbenchmark.

[148] Osmdroid, issue 53. https://code.google.com/p/osmdroid/issues/detail?id=53.

[149] Recycle-locator, issue 33. https://code.google.com/p/recycle-locator/issues/detail?id=33.

[150] Sofia public trasport, issue 76. https://github.com/ptanov/sofia-public-transport-navigator/issues/76.

[151] Ushahidi, issue 11. https://github.com/ushahidi/Ushahidi_Android/pull/11.

[152] Halachic prayer times. https://f-droid.org/wiki/page/net.sf.times.

[153] Sensor Tester. https://play.google.com/store/apps/details?id=com.dicotomica.sensortester.

[154] Omnidroid. https://f-droid.org/repository/browse/?fdid=edu.nyu.cs.omnidroid.app.

[155] Eclipse adt plugin. http://developer.android.com/tools/sdk/eclipse-adt.html.

[156] Get started with publishing. http://developer.android.com/distribute/googleplay/start.html.

[157] Statista. http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

[158] Monkeyrunner tool. http://developer.android.com/tools/help/MonkeyRunner.html.

[159] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, 2014.

[160] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[161] Greenery. https://github.com/ferno/greenery.

[162] Chessclock. https://f-droid.org/repository/browse/?fdfilter=chessclock.

[163] Sensorium. https://f-droid.org/wiki/page/at.univie.sensorium.

[164] App category. https://play.google.com/store/apps/category/APPLICATION?hl=en.

[165] Apache lucene core. https://lucene.apache.org/core/.

[166] Sensorium. https://play.google.com/store/apps/details?id=at.univie.sensorium&hl=en.

[167] Userhash. https://f-droid.org/repository/browse/?fdfilter=Userhash&fdid=com.threedlite.userhash.location.

[168] Sharemylocation. https://f-droid.org/repository/browse/?fdfilter=sharemyposition&fdid=net.sylvek.sharemyposition.

[169] Droidsat. https://f-droid.org/repository/browse/?fdfilter=droidsat&fdid=com.mkf.droidsat.

[170] Itlogger. https://f-droid.org/repository/browse/?fdfilter=itlogger&fdid=de.tui.itlogger.

[171] Heart rate monitor. https://f-droid.org/repository/browse/?fdfilter=heartrate&fdid=com.vanderbie.heart_rate_monitor.

[172] 0xbenchmark. https://f-droid.org/repository/browse/?fdid=org.zeroxlab.zeroxbenchmark.

[173] Ham. https://f-droid.org/repository/browse/?fdfilter=Ham&fdid=com.smerty.ham.

[174] Soot. http://sable.github.io/soot/.

[175] Monsoon power monitor. https://www.msoon.com/LabEquipment/PowerMonitor/.

[176] Sensorium repository - github. https://github.com/fmetzger/android-sensorium.

[177] Lide Zhang, B. Tiwana, R.P. Dick, Zhiyun Qian, Z.M. Mao, Zhaoguang Wang, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, 2010.

[178] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, 2014.

[179] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, 2010.

[180] Marius Marcu and Dacian Tudor. Energy consumption model for mobile wireless communication. In *Proceedings of the 9th ACM International Symposium on Mobility Management and Wireless Access*, MobiWac '11, 2011.

[181] Mian Dong and Lin Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 335–348, 2011.

[182] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering developers to estimate app energy consumption. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, pages 317–328, 2012.

[183] Alex Shye, Benjamin Scholbrock, and Gokhan Memik. Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, 2009.

[184] Denzil Ferreira, AnindK. Dey, and Vassilis Kostakos. Understanding human-smartphone concerns: A study of battery life. In *Pervasive Computing*, volume 6696, pages 19–33. Springer Berlin Heidelberg, 2011.

[185] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, 2011.

[186] Jason Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, WMCSA '99, pages 2–, 1999.

[187] A. Banerjee, L.K. Chong, S. Chattopadhyay, and A. Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *FSE*, 2014.

[188] Yepang Liu, Chang Xu, and S.C. Cheung. Where has my battery gone? finding sensor related energy black holes in smartphone applications. In *Pervasive Computing and Communications (PerCom), 2013 IEEE International Conference on*, 2013.

[189] Adam J. Oliner, Anand P. Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. Carat: Collaborative energy diagnosis for mobile devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 10:1–10:14.

[190] Lide Zhang, Mark S. Gordon, Robert P. Dick, Z. Morley Mao, Peter Dinda, and Lei Yang. Adel: An automatic detector of energy leaks for smartphone applications. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis*, CODES+ISSS '12, 2012.

[191] Abhijeet Banerjee, Hai-Feng Guo, and Abhik Roychoudhury. Debugging energy-efficiency related field failures in mobile apps. In *IEEE/ACM International Conference on Mobile Software Engineering and Systems*, MOBILESoft, 16, 2016.

[192] Irene Manotas, Lori Pollock, and James Clause. Seeds: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, 2014.

[193] A. Banerjee and A. Roychoudhury. Energy-aware design patterns for mobile application development (invited talk). In *Proceedings of the 2Nd International Workshop on Software Development Lifecycle for Mobile*, DeMobile 2014, 2014.

[194] Adel Noureddine and Ajitha Rajan. Optimising energy consumption of design patterns. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, 2015.

[195] B. R. Bruce, J. Petke, and M. Harman. Reducing energy consumption using genetic improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, 2015.

[196] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer. Post-compiler software optimization for reducing energy. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, 2014.

[197] William Martin, Mark Harman, Yue Jia, Federica Sarro, and Yuanyuan Zhang. The app sampling problem for app store mining. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, 2015.

[198] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, 2009.

[199] W. Eric Wong and Vidroha Debroy. Software fault localization. In *Encyclopedia of Software Engineering*, pages 1147–1156. 2010.

[200] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13, 2012.

[201] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811, 2013.

[202] Mytracks, issue 520. https://code.google.com/p/mytracks/issues/detail?id=520.

[203] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, 2009.

[204] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, 1982.

[205] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An empirical study of static call graph extractors. *ACM Trans. Softw. Eng. Methodol.*, 7(2), April 1998.

[206] J. M. Spivey. Fast, accurate call graph profiling. *Softw. Pract. Exper.*, 2004.

[207] Mikhail Dmitriev. Profiling java applications using code hotswapping and dynamic call graph revelation. In *Proceedings of the 4th International Workshop on Software and Performance*, WOSP '04, 2004.

[208] Csipsimple, issue 81. https://code.google.com/p/csipsimple/issues/detail?id=81.

[209] Sofia public transport, issue 38. https://github.com/ptanov/sofia-public-transport-navigator/issues/38.

[210] Adw-launcher-android, issue 202. https://code.google.com/p/adw-launcher-android/issues/detail?id=202.

[211] Google-voice-locaton, issue 4. https://code.google.com/p/android-google-voice-locations/issues/detail?id=4.

[212] Osmdroid isuue, 76. https://code.google.com/p/osmdroid/issues/detail?id=76.

[213] The Android Open Source Project. Adjusting the model to save battery and data exchange. http://developer.android.com/guide/topics/location/strategies.html.

[214] Omnidroid issue, 98. https://code.google.com/p/omnidroid/issues/detail?id=98.

[215] K9mail issue, 424. https://code.google.com/p/k9mail/issues/detail?id=424.

[216] Jdtcore. http://www.eclipse.org/jdt/core/.

[217] Ushahidi. https://www.ushahidi.com/.

[218] Shortyz. https://f-droid.org/wiki/page/com.totsp.crossword.shortyz.

[219] itlogger. https://f-droid.org/wiki/page/de.tui.itlogger.

[220] Mobiperf. https://f-droid.org/wiki/page/com.mobiperf.

[221] Strobe light. https://f-droid.org/wiki/page/com.stwalkerster.android.apps.strobelight.

[222] Userhash. https://f-droid.org/wiki/page/com.threedlite.userhash.location.

[223] Monsoon power monitor. https://www.msoon.com/LabEquipment/PowerMonitor/.

[224] Link to subject apps. http://www.comp.nus.edu.sg/~rpembed/energydebugger/subjectapps.xlsx.

[225] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, 2010.

[226] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, 2012.

[227] Lide Zhang, Mark S. Gordon, Robert P. Dick, Z. Morley Mao, Peter Dinda, and Lei Yang. Adel: An automatic detector of energy leaks for smartphone applications. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis*, CODES+ISSS '12, pages 363–372, 2012.

[228] Yepang Liu, Chang Xu, S.C. Cheung, and Jian Lu. Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *Software Engineering, IEEE Transactions on*, 40(9):911–940, 2014.

[229] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, 2011.

[230] App Revenue Statistics 2015. http://www.businessofapps.com/app-revenue-statistics/.

[231] Mobile Application Testing Market Boosted by Growing Demand for Automation. https://www.abiresearch.com/press/200-million-mobile-application-testing-market-boos/.

[232] PCI Security Standards Control. https://www.pcisecuritystandards.org/.

[233] JUnit. http://junit.org/.

[234] JMeter. http://jmeter.apache.org/.