Static Analysis Driven Performance and Energy Testing

Abhijeet Banerjee
National University of Singapore, Singapore
abhijeet@comp.nus.edu.sg

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 relationship of the software can be categorized as functional properties. On the other hand, properties that do not influence the input-output relationship 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.*

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging, C.3 [Special-Purpose and Application-Based Systems]

Keywords: Non-Functional Testing; Performance Stressing Test Input Generation; Energy-aware Test Generation

1. OVERVIEW

Real-time systems are a class of computer systems that must react to an external event within a pre-defined amount of time. Prime example of a real-time systems is the Anti-lock Braking System (ABS) used in contemporary automobiles. In such systems, missing a timing deadline may result in a system-failure, sometimes with fatal consequences. Due to the mission critical nature of such systems, they must be thoroughly tested for functional as well as non-functional properties. Verifying non-functional properties is often more complicated than verifying 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 coverage for a given non-functional property. Another related challenge is in choosing an 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 certain program property). However, due to absence of explicit annotation of non-functional properties in the application code, crafting such a coverage metric is often non-trivial.

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).

For software running on mobile platforms yet another kind of constraint arises due to the limited amount of onboard battery power. Common examples of such power constrained systems would be smarpthones and tablets. Unlike traditional desktop based systems, smartphone and tablets are often shipped with extensive run-time power management features. Such features enable the programmer to control the energy-consumption behaviour of the device from within the program (in order to reduce energy consumption). However, careless use of such features may make the program (and in some cases the device as a whole) hugely energy-inefficient. Note that the energy-inefficiencies in a program does not influence its functional behaviour directly, however such inefficiencies indirectly degrades the user-experience by reducing the time for which the device

FSE'14 , November 16–22, 2014, Hong Kong, China

can remain functional. This work then focuses on exploring and answering questions related to testing non-functional properties (mainly performance and energy consumption), of a program. More specifically, the work presented here is structure around the following three topics:

i. Representing non-functional properties in a fashion 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 automatic detection of inefficiencies related to extra functional properties

2. RELATED WORK

This section discusses some of the existing research that focuses on verifying non-functional properties of a program. Existing techniques have been divided in two categories: profiling based techniques and systematic testing techniques. Each category is further subdivided in two parts, one for techniques related to execution time analysis and another for techniques related to energy consumption analysis.

2.1 Profiling Based Techniques

Profiling can be described as a program 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 subsection we shall only highlight existing research techniques based on profiling.

On Execution Time:. Recent advances in profiling [1, 2] 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. Note that such cost functions are approximations and do not necessarily capture the actual cost of executing the program for a given input. In other words, such frameworks can potentially generate unsound results and therefore such methods are only suitable for average-case analysis.

On Energy Consumption: Previous works on energy-aware profiling [3] have shown poor energy behaviour of several smartphone applications. These works on profiling motivates the idea of energy-aware programming for smartphone applications. However, like any other profiling techniques, works proposed in [3] require specific input scenarios to execute the program on a device. A more recent work [4] has proposed a technique to relate power measurements with source lines of program. Such a technique also requires inputs to execute the program. Automatically finding such inputs is extremely non-trivial, as poor energy behaviour may be exposed only for a specific set of inputs.

2.2 Systematic Testing Techniques

Techniques in this category follow a methodical approach to testing. Such techniques often have a well defined coverage metric (related to the property of interest), which guides the exploration throughout the input space of the program.

On Execution Time: A number of existing works propose techniques for performance validation of a program. Such works include abstract interpretation based methods, such as [5], using which one can conservatively estimate the worst/best case execution time of the program. Methods similar to [5] can be very useful for hard real-time systems where absolute timing guarantees are required. However, such techniques perform the analysis irrespective of the program input and therefore such techniques cannot give an intuition of how a particular input affects the performance of the program. Other works, such as [6], use computation complexity as the metric for performance and develops testing techniques to highlight worst-case complexity scenarios. Such techniques provide some idea about the influence of an input on the program behaviour but they cannot be used to identify test-inputs that lead to bad performance for a specific micro-architectural components.

On Energy Consumption: Existing works related to the energy-consumption behaviour of a program can be divided into wide range of categories ranging from instruction level power models [7] to symbolic exploration based energy aware programming [8]. More recent works, such as [9], use data-flow analysis to detect energy-inefficiencies in Android applications. It is interesting to know that the domain of research related to energy consumption behaviour of a program has been continuously evolving so as to keep pace with contemporary hardware systems. Increasing use of mobile systems, (such as smartphones), has created a new host of energy-related issues to solve. It is also worthwhile to know that whereas earlier research works focused only on the energy consumption of CPU and its constituents, recent works encompass other axillary hardware components as well. This is necessary because in modern smartphones and similar mobile systems, power consumption from axillary hardware components (such as the GPS, Wifi and other I/O components) may rival the power consumption of the CPU itself. In the light of the recent developments, works such as [7] or [8] can be said to be preliminary in the sense that they only consider the CPU power consumption. In general, there is a lack of understanding as to what are the major reasons for energy-inefficiencies in such modern mobile platforms. Works such as [9] look at certain aspects of energy-inefficiency in smartphone applications. However, applicability of [9] would be limited because it is only applicable to command-line based applications whereas majority of smartphone applications are GUI based. Additionally, this technique cannot be used to generate test-inputs that trigger the malfunction in the application.

3. RESEARCH WORK

Our objective is to develop a technique that can be used for testing non-functional properties, such as performance and energy-consumption, of a program. Non-functional properties are seldom stated explicitly in the source code of the program. Therefore, such properties cannot be detected solely by monitoring the execution of the program. Additionally, the search space for real-life programs is often very large, therefore exhaustive exploration techniques are not scalable in practice. To overcome such challenges we propose a technique consisting of four key steps (Figure 1) (i) Identifying scenario for suboptimal non-functional behaviour (ii)

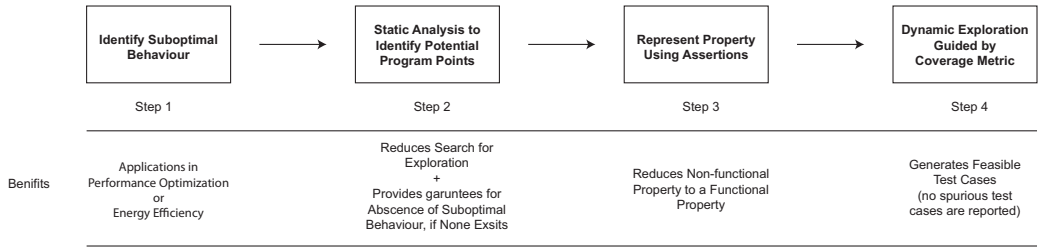

Figure 1: Key Steps In Our Test Generation Technique

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.

3.1 Identifying Scenarios for Suboptimal Non-functional Behaviour

Performance (or 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 (Publication A), 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 as well because such applications 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 are not well understood. Therefore, in one of our works (Publication B) 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 system calls provided by the operating system, therefore presence of such system 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 *system-call coverage* guided test generation framework to explore energy-inefficiencies in Android applications (Publication B). The framework automatically explores a given application while simultaneously analysing the energy consumption vs utilization behaviour 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.

3.2 Static Analysis to Identify Potential Program Points

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 [10]. 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).

3.3 Representation of Non-functional Properties as Assertions

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 extra 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 note 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.

3.4 Dynamic Exploration of Assertions

Existing dynamic exploration techniques, such as DART [11], can only check for validity of functional properties. This is no longer a problem because as a result of the previous step (instrumenting assertions) we have augmented the functional properties with the set of assertions capturing the non-functional properties. However, before we can apply the existing exploration techniques to test the validity of instrumented assertion, we need to address few issues. 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 performance optimization through input-sensitive cache locking. For improving energy efficiency, the results from our framework can be used for developing automated techniques for energy-aware code repair.

4. ACKNOWLEDGEMENT

This work was partially supported by Singapore MoE Tier2 grant MOE2013-T2-1-115.

5. LIST OF PUBLICATIONS

A. Static Analysis driven Cache Performance Testing Abhijeet Banerjee, Sudipta Chattopadhyay and Abhik Roychoudhury; RTSS, 2013.

B. Detecting Energy Bugs and Hotspots in Mobile Apps Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay and Abhik Roychoudhury; FSE, 2014 (*to appear*)

6. REFERENCES

[1] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. PLDI, 2012.

[2] E. Coppa, C. Demetrescu, and I. Finocchi. Input-sensitive profiling. PLDI, 2012.

[3] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. EuroSys, 2012.

[4] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating source line level energy information for android applications. ISSTA, 2013.

[5] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *RTS*, 2000.

[6] J. Burnim, S. Juvekar, and K. Sen. WISE: Automated test generation for worst-case complexity. ICSE, 2009.

[7] S. Malik V. Tiwari and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. ICCAD, 1994.

[8] T. Hönig, C. Eibel, R. Kapitza, and W. Schröder-Preikschat. SEEP: exploiting symbolic execution for energy-aware programming. HotPower, 2011.

[9] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. MobiSys, 2012.

[10] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. POPL, 1977.

[11] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. PLDI, 2005.