

SOFTWARE PRODUCTIVITY TOOL TO DETECT POWER INEFFICIENCIES CAUSED BY A SOFTWARE PROGRAM

PRIORITY CLAIM

[0001] This application claims priority from Singapore Patent Application No. 10201407015S filed on October 28, 2014.

TECHNICAL FIELD

[0002] The present invention generally relates to software productivity tools and systems for identifying power inefficiencies, and more particularly relates to a software productivity tool for developing a software program and a system for identifying power inefficiencies which are suitable for detecting energy bugs and energy hotspots.

BACKGROUND OF THE DISCLOSURE

[0003] Power inefficiencies have been extensively investigated for a number of devices and play a prominent role in mobile devices such as smartphones. Power inefficiencies (for example, in mobile applications) can broadly be categorized into energy bugs and energy hotspots. An energy bug can be described as a scenario where a malfunctioning application prevents the device from becoming idle, even after it has completed execution and there is no user activity. An energy hotspot can be described as a scenario where executing an application causes the smartphone to consume an abnormally high amount of battery power even though the utilization of its hardware resources is low.

[0004] Over recent years, the popularity of smartphones has dramatically increased. This has led to a widespread availability of smartphone applications. Further, 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. Also, modern smartphones are equipped with a wide range of sensors and input/output (I/O) components such as GPS and WiFi receivers as well as cameras. These I/O components allow developers to create a diverse set of applications. In spite of such high computation power and developer flexibility, the battery-life of these modern smartphones has improved only two or three times over the last decade or so. Since smartphones operate on a limited amount of battery power, it is important to develop tools and techniques that aid in power-efficient application development.

[0005] Thus, what is needed is a software productivity tool suitable for detecting energy inefficiencies and capable of detecting energy bugs and energy hotspots. Furthermore, other desirable features and characteristics will become apparent from the subsequent detailed description and the appended claims, taken in conjunction with the accompanying drawings and this background of the disclosure.

SUMMARY

[0006] Embodiments of the present invention provide a software productivity tool comprising a detection module and a guidance module.

[0007] According to a first aspect, there is provided a software productivity tool comprising: a detection module adapted to automatically detect a power inefficiency in a device comprising hardware components and to calculate an E/U ratio indicating power consumption to utilization of hardware components over a time

period, the power consumption and the utilization of the hardware components being caused by a software program, the E/U ratio used for detection of the power inefficiency in a device on which the software program is run; and a guidance module adapted to automatically generate at least some of a plurality of test inputs receivable by the software program and determine which of the generated test inputs will cause the power inefficiency in the device on which the software program is run.

[0008] According to a second aspect, there is provided a software productivity tool comprising: a detection module adapted to automatically detect a power inefficiency in a device comprising hardware components and to detect a presence of at least one event sequence likely to cause a power inefficiency in the device on which a software program is run, the at least one event sequence being triggered by at least one input receivable by the software program; and a guidance module adapted to automatically generate some of a plurality of test inputs receivable by the software program and determine which of the generated test inputs cause a power inefficiency in the device, the plurality of test inputs comprising the at least one input adapted to trigger the at least one event sequence.

[0009] According to a third aspect, there is provided a system for identification of power inefficiencies, the system comprising: a portable device having an energy storage device for supplying power thereto, the portable device being configured to run software programs thereon; and a software productivity tool coupled to the portable device for identifying power inefficiencies in operation of one or more of the software programs by the portable device, the software productivity tool comprising: a detection module adapted to automatically detect a power inefficiency in a device comprising hardware components and to detect a presence of an event sequence likely to cause a power inefficiency in the portable device on which the one or more of the software programs are run, the event sequence being triggered

by an input receivable by the one or more of the software programs; and a guidance module adapted to automatically generate some of a plurality of test inputs receivable by the one or more of the software programs and determine which of the generated test inputs are likely to cause the power inefficiencies in the portable device, the plurality of test inputs comprising the at least one input adapted to trigger the event sequence.

BRIEF DESCRIPTION OF THE DRAWINGS

[0010] The accompanying figures, where like reference numerals refer to identical or functionally similar elements throughout the separate views and which together with the detailed description below are incorporated in and form part of the specification, serve to illustrate various embodiments and to explain various principles and advantages in accordance with a present invention, by way of non-limiting example only, wherein:

[0011] FIG. 1 depicts a table of a classification of exemplary energy bugs and energy hotspots in accordance with a present embodiment.

[0012] FIG. 2 depicts a diagram of an exemplary lifecycle of a software program activity on a device in accordance with the present embodiment.

[0013] FIG. 3 depicts a snippet of application code of an exemplary software program with a potential energy bug in accordance with the present embodiment.

[0014] FIG. 4, comprising FIGs. 4A and 4B, depicts snippets of application code of exemplary software programs in accordance with the present embodiment, wherein FIG. 4A depicts a snippet of application code comprising an energy hotspot due to disaggregated communication and FIG. 4B depicts the snippet of application code without the energy hotspot.

[0015] FIG. 5 depicts a bar graph of a power profile for a smartphone portable device in accordance with the present embodiment.

[0016] FIG. 6 depicts a block diagram of a system for identifying power inefficiencies in accordance with the present embodiment.

[0017] FIG. 7, comprising FIGs. 7A and 7B, depicts event flow graphs in accordance with the present embodiment, wherein FIG. 7A depicts an event flow graph of before activation of an eject button and FIG. 7B depicts an event flow graph of after activation of the eject button.

[0018] FIG. 8 depicts a table categorizing system calls in accordance with the present embodiment.

[0019] FIG. 9 depicts graphs of exemplary energy-consumption to utilization (E/U) traces in accordance with the present embodiment.

[0020] FIG. 10 depicts a flowchart of operation of the system of the software productivity tool of FIG. 6 in accordance with the present embodiment.

[0021] FIG. 11 depicts a block diagram of an experimental setup for a system for identification of power inefficiencies utilizing the software productivity tool of FIG. 6 in accordance with the present embodiment.

[0022] FIG. 12 depicts a tire pie chart diagram categorizing the test software programs used in the system of FIG. 11 to identify power inefficiencies in accordance with the present embodiment.

[0023] FIG. 13 depicts a table of statistics for energy hotspots and energy bugs found by the system of FIG. 11 in the test software programs of FIG. 12 in accordance with the present embodiment.

[0024] FIG. 14, comprising FIGs. 14A and 14B, depict graphs of energy traces of an event trace for a first one of the test software programs of FIG. 12 in accordance with the present embodiment.

[0025] FIG. 15, comprising FIGs. 15A and 15B, depict graphs of energy traces of an event trace for a first one of the test software programs of FIG. 12 in accordance with the present embodiment.

[0026] And FIG. 16 depicts a table of coverage statistics from the test software programs of FIG. 12 in accordance with the present embodiment.

[0027] Skilled artisans will appreciate that elements in the figures are illustrated for simplicity and clarity and have not necessarily been depicted to scale. For example, the size of some of the components in the block diagram of FIG. 11 may not be accurate in respect to other components to help to improve understanding of the system for identifying power inefficiencies in accordance with the present embodiment.

DETAILED DESCRIPTION

[0028] The following detailed description is merely exemplary in nature and is not intended to limit the invention or the application and uses of the invention. Furthermore, there is no intention to be bound by any theory presented in the preceding background of the invention or the following detailed description. It is the intent of this invention to present a software productivity tool and a system which uses the tool to identify power inefficiencies. The tool detects energy hotspots and energy bugs in software applications by generating test inputs that are likely to capture energy hotspots and/or energy bugs. Each test input captures, for example, a sequence of user interactions that lead to energy hotspots and/or energy bugs in the application.

[0029] In accordance with a present embodiment, energy hotspots, energy bugs, utilization (U), an energy-consumption to utilization (E/U) ratio, an event flow graph (EFG), and an event trace are defined as follows. An energy hotspot is described as a

scenario where executing an application causes a device to consume an abnormally high amount of power even though the utilization of its hardware resources is low. An energy bug is described as a scenario where a malfunctioning application prevents a device from becoming idle even after it has completed execution and there is no user activity. Utilization (U) is defined as the weighted sum of utilization rates of all major power consuming hardware components in a device, over a given period of time. The energy-consumption to utilization (E/U) ratio is defined as the measure of energy inefficiency of an application for a given time period. An Event Flow Graph (EFG) is a directed graph, capturing all possible user event sequences that might be executed via a graphical user interface (GUI), where nodes of an EFG represent GUI events and a directed edge between two EFG nodes X and Y represents that a GUI event Y follows a GUI event X . An event trace is defined as a path of arbitrary length in the EFG which starts from an event in the root screen of the respective application.

[0030] 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. 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 input/output (I/O) components. These I/O components allow developers to create a diverse set of applications.

[0031] Smartphones, however, are portable devices. Thus, 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. Yet, the battery-life of these modern smartphones has improved only two or three times. High computational power coupled with small battery capacity and application development in an energy-oblivious fashion has led to a shortened battery life and an unsatisfied user base.

[0032] Energy inefficiencies in smartphone applications can broadly be categorized into energy hotspots and energy bugs. Referring to FIG. 1, a table 100 lists different types of energy bugs 102 and energy hotspots 104 that can be found in smartphone software programs (i.e., smartphone applications). Many smartphones are designed to operate at different power states to prolong battery life. However, as shown in the table 100, malfunctioning applications may lead to inappropriate power states, such as energy hungry GPS/sensor updates and non-idle power states operating in the absence of user activity. For hardware resources 110, energy bugs 102 may include resource leaks 112 and energy hotspots 104 may include suboptimal resource binding 114. For sleep-state transition heuristics 120, energy bugs 102 may include wakelock bugs 122 and energy hotspots 104 may include tail-energy hotspots 124. For background services 130, energy bugs 102 may include vacuous background services 132 and energy hotspots 104 may include expensive background services 134. And for defective functionalities 140, energy bugs 102 may include immortality bugs 142 and energy hotspots 104 may include loop-energy hotspots 144. Most of these power inefficiencies appear when the application does not access 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 to power inefficiencies in application code of software programs. Presence of such power inefficiencies in the application code can be identified in accordance with the present embodiment.

[0033] In addition to the sources of energy consumption in smartphone applications shown in the table 100, it is known that I/O components are primary sources of energy consumption in a smartphone. The I/O components are usually accessed in application code via system calls. In addition, power management functionality such as Wakelocks, background services and other hardware resources of a device can only be accessed through a set of system calls as shown in the table 100. In summary, most of the energy hotspots/bugs classified in the table 100 are exposed via the invocation of system call(s). Therefore, a software productivity tool in accordance with the present embodiment systematically generates user interaction scenarios which potentially invoke such system calls.

[0034] The software productivity tool is also designed in accordance with the present embodiment to systematically traverse an event flow graph (EFG). An EFG is an abstraction to capture a set of possible user interaction sequences. Each node in an EFG captures a specific user interaction such as a user touching a button on a 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, appropriate EFG traces are generated in accordance with the present embodiment which are likely to lead to undesirable energy consumption. To accomplish this, two strategies are employed.

[0035] From the information of table 100, EFG traces are selected that invoke system calls that might be responsible for irregular energy consumption. In addition, if an energy hotspot/bug is detected after executing an EFG trace, in accordance with the present embodiment the sequence of system calls responsible for such irregular energy behaviour is recorded. Subsequently, unexplored EFG traces that may invoke

a similar sequence of system calls are prioritized with a goal to uncover as many energy hotspots/bugs as possible in a limited time budget.

[0036] In accordance with the present embodiment, a test input (i.e. a user interaction scenario) is executed on a portable device (e.g., a smartphone) to detect energy hotspots/ bugs while simultaneously measuring the power consumption via a power meter. The energy bug is detected in a specific trace by measuring statistical dissimilarities in a power consumption trace of the device, specifically, before and after executing the respective application. As power consumption of idle devices should be similar, a statistical dissimilarity indicates an energy bug.

[0037] To detect an energy hotspot, an anomaly detection technique is employed to locate anomalous power consumption patterns. Once the process of detecting hotspots/bugs in a power consumption trace is completed, a different user interaction scenario is generated using the EFG. In accordance with the present embodiment, the test generation process continues until the time budget expires or all event traces invoking system calls have been explored. As the system calls are the potential locations to cause irregular energy behaviour, the quality of the software productivity tool in accordance with the present embodiment is dependent on the coverage of system calls in the application.

[0038] The present embodiment is tested using Android™ software programs operating on smartphones. Android™ is an open-source operating system (OS) designed for mobile devices such as smartphones and 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).

[0039] The user interaction interface of an Android application is referred to as an activity. Referring to FIG. 2, a diagram 200 depicts an exemplary lifecycle of an Android activity. An activity can be in one of the seven stages during its lifecycle between launch 202 of the activity and shutdown 204 of the activity. Usually, all the set-up tasks (such as acquiring resources and starting background services) take place in four stages of the activity, namely onCreate 206, onStart 208, onResume 210 and onRestart 212 after which the activity is running 213. Similarly, all the tear-down tasks (such as releasing resources and stopping background services) take place in three stages, namely onPause 214, onStop 216 and onDestroy 218. An activity may be killed 220 by a user navigating to another activity or in response to an activity with a higher priority needing memory. Yet, some real-life applications do not follow the ideal set-up and tear-down scenarios as depicted in the diagram 200. Such applications may, for example, contain energy bugs. In addition, most real-life applications have a huge number of feasible user interaction scenarios (due to complex GUIs). As a result, it is impossible for a software developer to test an application for all possible scenarios.

[0040] FIG. 3 depicts a snippet 300 of application code that has a potential energy bug. The application code is supposed to start a location update background service a line 310 in the onCreate method. Subsequently, it performs some operation with list data at line 320. When the user stops the application, the location update service is removed at line 330 in the onStop method. However, if there is an exception before the line 330 (for instance, due to line 340), the location update service is never stopped resulting in an energy bug. The application code snippet 300 shows one possible scenario, vacuous background services 132 which can lead to an energy bug.

[0041] Referring to FIG. 4A, a code snippet 400 depicts an example of application code with energy hotspots due to disaggregated network activities (an example of a tail-energy hotspot 124). Observe that in the application code snippet 400, network related code at line 402 is interleaved with CPU-intensive code at line 404 within the same loop 406. Such an interleaving causes energy-inefficiencies due to tail-energy. Tail-energy behaviour has been observed for network hardware components in smartphones such as 3G, GSM and WiFi receivers. Tail-energy behaviour has also been observed in hardware components such as storage disks and GPS receivers. In order to reduce energy-loss due to tail-energy, the network related code in the application code snippet 400 can be aggregated as shown in a snippet 410 of application code in FIG. 4B.

[0042] While power consumption ratings on portable devices can be obtained by performing empirical experiments based on provided guidelines (e.g., in regards to Android smartphones, such guidelines are provided on an Android developer web page), such power consumption ratings can also be obtained from the average power consumption ratings for the hardware components in the device provided in a XML file shipped with the device. The data contained in the XML file is provided by the device manufacturer and may be used to show battery related statistics. However, note that the data in the 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. An example of power profile data from an XML file for a smartphone is shown in a bar graph 500 in FIG. 5. The bar graph 500 shows power consumption by operation of a smartphone touchscreen 502, a Bluetooth receiver 504, a GPS receiver 506, a WiFi transceiver 508, a 2G/3G transceiver 510 and a CPU 512, the CPU 512 operating at various frequencies.

[0043] Referring to FIG. 6, a block diagram 600 of a system for identifying power inefficiencies in accordance with the present embodiment includes a portable device 602, a power meter 604 and a software productivity tool including a detection module 606 and a guidance module 608. The portable device 602 is a smartphone which includes an energy storage device for supplying power thereto and is configured to run software programs thereon. The power meter 604 is coupled to the portable device 602 for measuring power consumption when the software programs are run on the portable device 602. The detection module 606 includes a hotspot/bug detector 610 which is coupled to the portable device 602 to receive hardware component utilization information therefrom and is coupled to the power meter 604 to receive an energy consumption trace therefrom in order to detect within at least one event sequence an energy hotspot and/or an energy bug.

[0044] The guidance module 608 includes a guidance component 612 coupled to the hotspot/bug detector 610 to receive energy hotspot/bug information therefrom to select subsequent event traces for providing to the portable device in accordance with the present embodiment. Thus, the guidance component 612 is adapted to generate inputs from data stored in a database 614, the inputs receivable by the software programs of the portable device 602, the guidance component 612 determining which of the generated inputs are likely to cause power inefficiencies in the portable device 602 from the energy hotspot/bug information and the inputs being provided as event traces to the software programs to trigger an event sequence in the software program. In this manner, the guidance component 612 provides selected event traces that are more likely to uncover energy hotspots/bugs to the software programs for guided exploration thereof. The process of selection, execution and detection continues until

the given time budget has expired or all event traces invoking system calls have been explored.

[0045] An event flow graph (EFG) extraction module 616 receives event flow data from the software programs/applications in the portable device 602 and provides EFG information to an event trace generator 618 for generating event traces that lead to energy hotspots/bugs and storing the event traces in the data base 614 for later reporting to software developers and for further investigation.

[0046] For the hotspot/bug detector 610 to detect a hotspot/bug, the power consumption of the application for a given event trace is measured by the power meter 604 and is provided as an energy consumption trace to the hotspot/bug detector 610. However, it is difficult to detect a hotspot/bug in an application solely by analyzing its power consumption trace. For instance, where two programs P_1 and P_2 have similar power consumption traces and program P_1 has a much higher utilization of system resources as compared to P_2 , program P_1 is more energy-efficient than program P_2 . Therefore, to accurately detect energy inefficiencies, an appropriate metric for system-resource utilization is defined in accordance with the present embodiment.

[0047] 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 the 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. Thus, hardware component utilization $Load_x$ is provided to the hotspot/bug detector 610 for energy hotspot/bug detection.

[0048] Based on the power profile shown in the bar graph 500, major power consuming components in smartphone portable device 602 are the screen 502, the WiFi transceiver 508, radio transceiver 510, the GPS receiver 506 and the CPU 512. Therefore, for a given time interval, the utilization of system resources can be computed by Equation (1) using Equations (2), (3) and (4)

$$\text{Utilization} = U_{Screen} + U_{CPU} + U_{WiFi} + U_{Radio} + U_{GPS} \quad (1)$$

$$U_{CPU} = \begin{cases} W_{CPU_{320}} \times Load_{CPU}, & \text{if CPU is operating at 320MHz} \\ W_{CPU_{480}} \times Load_{CPU}, & \text{if CPU is operating at 480MHz} \\ W_{CPU_{600}} \times Load_{CPU}, & \text{if CPU is operating at 600MHz} \\ W_{CPU_{800}} \times Load_{CPU}, & \text{if CPU is operating at 800MHz} \end{cases} \quad (2)$$

$$U_{Screen} = \begin{cases} W_{Screen_{ON}} \times Load_{Screen}, & \text{if screen on} \\ W_{Screen_{FULL}} \times Load_{Screen}, & \text{if at full brightness} \end{cases} \quad (3)$$

$$U_x = W_x \times Load_x, \quad x \in \{WiFi, Radio, GPS\} \quad (4)$$

[0049] In Equation (1), 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 (see the bar graph 500 in FIG. 5). Specifically, the value of W_x is normalized such that W_x for the most power consuming component is 1 (e.g., $Screen_{FULL}$ as shown in the bar graph 500). Note that Equation (1) does not include the Bluetooth receiver 504. This is because the Bluetooth receiver 504 has a very low power consumption as compared to the other hardware components. However, if required, Equation (1) can easily be extended to accommodate the Bluetooth receiver 504 as well. Using the new metric of utilization (U), the magnitude of energy-inefficiency can now be computed.

[0050] If the energy consumption to utilization (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. A high E/U ratio during the execution of an application indicates the presence of an energy hotspot. On the contrary, a persistently high E/U ratio even after the application has completed execution indicates the presence an energy bug.

[0051] Event traces based on an event flow graph (EFG) of a software program are utilized to reveal hotspots and bugs. In accordance with the present embodiment, the EFG extraction module 616 uses a modification of a conventional Dynodroid™ tool to generate the EFG while the software program (i.e., the application) is running on the portable device 602. The event trace generator 618 receives the EFG from the EFG extraction module 616 and generates event sequences up to maximum length k and stores them in the database 614. After the event traces have been generated, the guidance component 612 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. The guidance component 612 accomplishes this by selecting event traces based on the number of invoked system calls and guidance heuristic. The guidance heuristic gathers information from previously detected hotspots/bugs, specifically the sequence of system calls which are likely to lead to energy inefficiencies. Subsequently, the selection process of the guidance component 612 is biased towards event traces invoking a similar sequence of such system calls. This process of selection, execution and detection continues by the guidance component 612 until the time budget has expired or all event traces invoking system calls have been explored. Thereafter, event traces that lead to energy hotspots/bugs may be reported to software developers for further investigation.

[0052] Thus it can be seen that operation in accordance with the present embodiment includes two steps: a preprocessing step that preprocesses the application under test by the EFG extraction module 616 and the event trace generator 618 to build within the database 614 a plurality of possible event traces, and a test generation step performed by the guidance module 612 and the hotspot/bug detector 610 using the event traces generated during the preprocessing.

[0053] Preprocessing of an application can be divided into three steps: EFG extraction by the EFG extraction module 616, event trace generation by the event trace generator 618 and extraction of system calls sequence for each event trace. Note that this preprocessing step is performed only once for each application. The generated EFG and event traces are stored in the database 614 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 without repeating the preprocessing step.

[0054] The EFG is built based on a conventional user interface (UI) model where two third-party tools, Hierarchy Viewer™ and Dynodroid™, are used for the purpose of EFG construction. The Hierarchy Viewer provides information about the UI elements of the application under execution and the Dynodroid is used to automatically explore the event sequences. Note that Dynodroid does not generate the EFG by itself, therefore the modified Dynodroid source code is used to build the EFG. The EFG is constructed gradually each time the modified Dynodroid interacts with the application.

[0055] Referring to FIG. 7, including FIGs. 7A and 7B, flow graphs 700, 702 shows how the EFG is gradually built as Dynodroid performs the exploration of event sequences in accordance with the present embodiment among graphical user interface

(GUI) states PlayButton 704, StopButton 706, SkipButton 708, RewindButton 710, PauseButton 712 and EjectButton 714. As defined above, an EFG is a directed graph, capturing all possible user event sequences that might be executed via a GUI, where nodes of an EFG represent GUI events and a directed edge between two EFG nodes X and Y represents that a GUI event Y follows a GUI event X. It is worthwhile to note that Dynodroid does not guarantee to reach all GUI states during exploration. Therefore, the constructed EFG is in fact a partial EFG of the entire application. However, it has been observed during operation in accordance with the present embodiment that the generated EFGs cover most of the GUI elements for the tested applications.

[0056] Each EFG is primarily used to generate a set of event traces. Note that each application has a start GUI screen. This GUI screen is called the root screen and is presented to the user when an application is launched. Thus, 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, as defined above an event trace is a path of arbitrary length in the EFG which starts from an event in the root screen of the respective application .

[0057] Based on our EFG, we generate a complete set of event traces up to length k . These event traces are stored in the database 614 for further analysis during test generation. Referring to FIG. 7B, the flow graph 702 shows the partial EFG of an application which has additional GUI states BackButton 720, Button2 722 and Button1 724 flowing from the EjectButton 714 node. The PlayButton 704 node captures the root screen of the same application. An example event trace of length three would be PlayButton 704 $\rightarrow$ StopButton 706 $\rightarrow$ PlayButton 704 or SkipButton

708 → EjectButton 714 → BackButton 720. Note that events PlayButton 704 and SkipButton 708 correspond to different events in the root screen of the application.

[0058] Input/output (I/O) hardware components are one of the major sources of energy consumption in smartphones (see the graph 500 of FIG. 5). In general, for modern smartphones the major power consuming components are the screen, the CPU, the WiFi transceiver, the radio (e.g., 2G/3G) transceiver, the GPS receiver, a secure digital card (SD Card) interface, a camera and audio hardware. These hardware components (except for the CPU) can only be accessed via a set of system calls (e.g., application program interfaces (APIs)) provided by a software development kit (SDK) such as the Android SDK. Referring to FIG. 8, a table 800 categorizing a pool of systems calls based on their functionalities. It is worthwhile to note that such a pool is constructed only once and it needs to be updated only if the SDK framework changes.

[0059] The application code locations which invoke any system calls from the constructed pool are instrumented. The instrumented code runs in an emulator on a computer to collect the system call traces during the execution of an event trace. The instrumented code on the emulator is executed and the system calls invoked for each event trace are recorded. These system calls are annotated with the EFG node corresponding to the triggered event. Thus, for each event trace generated from the EFG, the respective system call trace can be generated. It is important to note that the event traces are executed on the portable device 602 (i.e., the smartphone) as well as in the emulator. The instrumented application runs on the emulator whereas the instrumentation-free application runs on the smartphone. Thus, the instrumentation does not influence the energy consumption behaviour of the application.

[0060] Energy hotspots/bugs are those regions of code that lead to high E/U ratio. To detect energy hotspots during an event trace T , the E/U ratio trace (E/U_T), during the execution of T is obtained. As shown in the E/U_T vs time graph 900 of FIG. 9, E/U_T is divided into four different stages: pre-execution stage (PRE) 902, execution stage (EXC) 904, recovery stage (REC) 906 and post execution stage (POST) 908 where a first E/U_T 910 shows a E/U ratio trace with an energy hotspot, a second E/U_T 920 shows a E/U ratio trace with an energy bug, and a third E/U_T 930 shows a E/U ratio trace with no energy hotspot/bug.

[0061] The rationale for dividing E/U_T trace into four stages is as follows: in the PRE stage 902 the execution of the event trace T has not started yet. Therefore, the PRE stage 902 records the idle-behaviour (low-power state) of the device. Similarly, in the POST stage 908, the device has completed execution of T and so in an ideal scenario the device would have gone back to its idle-behaviour during the POST stage 908. The execution stage 904, 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—this is the recovery stage 906.

[0062] To detect the presence (or absence) of an energy bug we compare the E/U_T values in the PRE stage 902 and the POST stage 908 using statistical methods. If the dissimilarity between E/U_T values in the PRE stage 902 and the POST stage 908 is more than a predefined threshold (e.g., a predefined threshold of 50%), an energy bug is flagged (i.e., execution of T changed the idle-behaviour of the device). This can be seen in the second E/U_T 920.

[0063] Compared to detection of bugs, detection of hotspots is much trickier. Hotspots may appear only during the execution of an event trace (i.e., during the EXC

stage 906) or just after the execution of an event trace (i.e., during the REC stage 906). Note that E/U_T values obtained in the EXC stage 904 and the REC stage 906 may substantially vary for different event traces. In addition, different executions of the same event trace may show different E/U_T values in the EXC stage 904 or the REC stage 906, due to different hardware states. Therefore, a clear definition of energy hotspots is needed to detect them automatically. Abnormally high energy wastage during the execution of an event trace is a suitable indicator of energy hotspots. To detect such unusual energy behaviour, connections from data mining and classification techniques can be used and the problem of detecting unusual energy behaviour is similar to detecting unusual subsequences in time-series data. Thus, an anomaly detection technique that computes discords in a time-series data are used for hotspot detection in accordance with the present embodiment. Discords are subsequences in a time-series data that are maximally different from the rest of the time-series. Discords are detected within the E/U_T values from the EXC stage 904 and REC stage 906 to highlight subsequences in the E/U_T that are abnormally different from the rest of the subsequences in the EXC stage 904 and REC stage 906. Additionally, the anomaly detection also points out the magnitude of each computed discord. For instance, in first E/U_T 910, discord D_1 912 has a higher anomaly magnitude than the discord D_2 914. These magnitudes are extremely helpful because the computed energy hotspots can be ranked based on their magnitude for prioritization of energy hotspot removal. In accordance with the present embodiment, JMotif™ is used as the anomaly detector. JMotif is an off-the-shelf data mining library and it includes the implementation of finding discords in a time-series data.

[0064] Guidance heuristics are used for test generation to select unexplored event traces that have 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 systems calls in the event trace, (b) similarity to previously explored hotspot/bug revealing event traces, and (c) starvation of event traces due to unexplored system calls.

[0065] Since the major power consuming hardware components in smartphones can be accessed through a set of system calls, the presence of system calls that activate (or deactivate) such hardware components can be used for guiding test generation. At the beginning of a test generation process, all event traces are ranked according to the number of such system calls they can invoke. In subsequent iterations, the guidance component 612 of the guidance module 608 becomes more intelligent by learning specific system call subsequences 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 systems calls they can invoke. Note that such a comparison is perfectly feasible, as the system call trace for each event trace is extracted during the preprocessing stage. In accordance with the present embodiment, a similarity between two system call traces is compared using the Jaro Winkler Distance™ algorithm.

[0066] The third parameter, guidance by starvation, aims to cover as many system calls as possible during exploration. Since the first two parameters are based on the number of system calls and the exploration history, it is possible that the guidance heuristics may ignore several unexplored system calls. This leads to starvation, where a set of system calls will never be explored by the test generation process. Such

starvation is undesirable, as unexplored system calls may potentially expose new energy hotspots/bugs. Therefore, to ensure a fair coverage of all the system calls invoked by an application, a guidance parameter is added to deal with the problem of starvation. Essentially, guidance by starvation ranks all unexplored event traces by the ratio of the number of unexplored system calls in an event trace to the total number of system calls in all event traces.

[0067] Referring to FIG. 10, a flowchart 1000 depicts an algorithm for test generation by the guidance component 612 in accordance with the present embodiment. The primary objective of the algorithm is to uncover as many energy hotspots/bugs as possible in an application within a given time budget. A database of unexplored events is inputted 1002 into the algorithm. In accordance with the present embodiment, the database of an Android application's event traces is generated from the EFG of the application by the event trace generator 614 and stored in the data base 614. The guidance component 612 systematically executes the event traces from the database 614 on the smartphone.

[0068] The guidance parameters as modified at step 1004 are used for the test generation. An event trace, E , is selected 1006 from the data base 614, a monitoring device (e.g., the power meter 604) is started 1008 and the event trace E is executed 1010 on the smartphone after which the monitoring device is stopped 1012. The E/U data from the detection module 606 for each execution is analyzed 1014 by the guidance component 612 for presence of energy hotspots/bugs. If the E/U data contains an energy hotspot/bug 1016, the test suite is updated 1018. With or without the test suite being updated 1018, the database is updated 1020 by re-ranking all unexplored event traces. The exploration continues until the allocated time budget has expired 1022 after which the test suite 1024 is complete for storing and/or

reporting. Thus, on completion, the framework reports a set of event traces (i.e., the test suite), each of which leads to an energy hotspot/bug when executed on the device.

[0069] The two most important components, i.e., the guidance heuristics for test generation and the technique of hotspot/bug detection, have been discussed in the preceding paragraphs. There is however, one more component of the framework that must be explained. At step 1004, the guidance parameters, α , β and γ are refined which means that the reliance (or the weight) of the various guidance parameters are refined. Recall that the guidance heuristics is based on three parameters: guidance by number of system calls (corresponding weight would be α), guidance by exploration history (corresponding weight would be β), and guidance by starvation of system calls (corresponding weight would be γ). Assume that, for a given event trace E , guidance by number of system 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 . Equation (5) is used to obtain a single score S_E for an unexplored event trace E .

$$S_E = \alpha \times G_n + \beta \times G_h + \gamma \times G_s \quad (5)$$

where $\alpha + \beta + \gamma = 1$. In Equation (5), α , β and γ are three tuneable factors which drive the priorities of different guidance parameters. At the start of the algorithm of the flowchart 1000 without any knowledge about likely energy hotspots/bugs, α is initialized to 1 and both β and γ are initialized to 0. During each successive 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 system calls. Specifically, during each iteration the value of α is decreased at step 1004 by a fixed amount Δ ($0 < \Delta < 1$). If an energy hotspot was found 1016 in the previous iteration, the value of β is increased at step 1004 to $\beta + \Delta$. This refinement of β at step 1004 finds energy hotspots/bugs that had similar system call sub-sequences as previously found hotspots/bugs. The value of β

continues to increase in each iteration at step 1004 as long as energy hotspots/bugs are found or until the value of β reaches 1. If no energy hotspots/bugs are found in an iteration, the weight of γ is increased to $\gamma + \Delta$ to reach previously unexplored system calls. This assignment of extra weight Δ is taken from α , if $\alpha \geq \Delta$. Otherwise, the value of β is modified to $\beta - \Delta$ to decrease the priority of the execution history.

[0070] Operation in accordance with the present embodiment was evaluated to answer three questions: What is the efficacy of the present embodiment in uncovering energy bugs and hotspots in real-world applications? How can an application developer benefit from the reports generated in accordance with the present embodiment? And is guidance based on system call coverage a more appropriate metric than code coverage for uncovering energy bugs and hotspots?

[0071] Referring to FIG. 11, a block diagram 1100 depicts an experimental setup for a system for identification of power inefficiencies in accordance with the present embodiment. An Optimus™ L3 smartphone 1102 manufactured by LG Electronics of Seoul, Korea was used as the portable device on which to run the subject software programs. The device 1102 has a single core processor and features standard I/O components such as GPS, WiFi, 3G and Bluetooth. The device 1102 uses an Android 2.3.3 (Gingerbread™) operating system (OS). To monitor energy consumption of the smartphone 1102, a WT210 digital power meter 1104 manufactured by Yokogawa Electric Corporation of Tokyo, Japan was used for precise power measurement. The power meter 1104 was coupled to a battery 1106 providing power to the device 1102 for current and voltage measurements. A special apparatus housed the battery 1106 to allow the voltage and the current flowing through the battery to be measured without any distortion.

[0072] The energy-testing software productivity tool ran on a desktop personal computer (PC) 1108 that had an Intel® Core i5 processor and 4 GB RAM. The OS used on the desktop PC 1108 was Windows 7®. The desktop PC 1108 was coupled to the power meter 1104 via a serial port 1110 to receive output therefrom. The desktop PC 1108 was also coupled to the smartphone 1102 via an Android debug bridge 1112 for sending event traces to the smartphone 1102 and for obtaining hardware component utilization data therefrom. The software productivity tool in accordance with the present embodiment ran on the desktop PC 1108, which also served as the global clock. All measurements from the power-meter 1104 reporting power consumption data and the smartphone 1102 reporting utilization data were collected by the desktop PC and each reading was recorded with a timestamp generated on the desktop PC. Since all of the timestamps were generated by the clock on the desktop PC, the data from the power meter 1104 and the smartphone 1102 can be synchronized by the timestamps.

[0073] A tire pie chart diagram 1200 in FIG. 12 depicts a categorization of the subject software programs used in the system of FIG. 11 to identify power inefficiencies in accordance with the present embodiment. The subject software programs are all available on either the Google Play™ store or the F-droid™ repository. A total of thirty different Android applications from different categories (e.g., tools, productivity, transportation) as shown in the diagram 1200 were used. The subject programs are diverse in terms of APK (Android application package file) size with the largest application tested was 8.0MB in size while the smallest application tested was 22KB in size. The average APK size of the subject programs was 1.1MB.

[0074] The subject programs also had varying GUI complexity where GUI complexity of an application is measured 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 (e.g., to a length of four), the selected subject software programs contained between 26 to 2,800 feasible event traces.

[0075] Note that in accordance with the present embodiment, the source code of an application is not required to detect energy hotspots/bugs. However, the source code is required to obtain code coverage metrics and for debugging purposes. Therefore, only open-source applications were considered where the source code is needed to perform the evaluation. The lines of code for the open-source applications used in accordance with the present embodiment varied from 448 to 11,612, with an average of 4010 lines of code per application.

[0076] The efficacy of the present embodiment in quickly uncovering energy bugs and hotspots in real world applications was determined by evaluating the subject software programs using a time budget of twenty minutes. Additionally, exploration for event traces was limited up to a length of four. A summary of the bugs and hotspots reported by experiments using the setup of FIG. 11 and the software programs of FIG. 12 is listed in a table 1300 of FIG. 13.

[0077] Operation in accordance with the present embodiment reported energy bugs for ten of the thirty subject software programs. Energy hotspots were also reported for three of the subject software programs. Note that the hotspot detection technique in accordance with the present embodiment is based on an anomaly detection method and, 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, the

respective event trace was manually executed on the device 1102 to observe whether the same energy hotspot could be replicated. The result of the manual validation as shown in the table 1300 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 four (as shown in the table 1300). In spite of this large number of event traces, the software productivity tool in accordance with the present embodiment can advantageously 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 system call usage.

[0078] After analyzing an application, a test report is generated in accordance with the present embodiment. This report advantageously 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 in accordance with the present embodiment. Each test case includes (i) a MonkeyRunner™ script for automatic execution of events that lead to the energy issue, (ii) energy trace patterns, (iii) details of the energy issue (e.g., magnitude of an energy hotspot), and (iv) the set of system calls invoked.

[0079] From the report, a software developer may advantageously 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.

[0080] While executing an event trace T that exposes an energy hotspot, 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 advantageously reduced to the code fragments that were executed before E was triggered. This helps a software developer in fixing the reported energy issues. Below, case studies are discussed on two of the analyzed applications (an energy bug was reported for one and a hotspot for the other) to demonstrate how a developer can advantageously utilize the generated reports to debug and fix energy issues in applications.

[0081] The first case study involves an Aripuca™ GPS Tracker wherein two event traces with energy bugs were discovered by operation in accordance with the present embodiment. The energy consumption pattern for such an event trace is shown in a graph 1400 in FIG. 14A. As shown in the graph 1400, the energy consumption in the POST stage 1408 is not similar to the energy consumption in the PRE stage 1402 indicating an energy bug. Therefore, the device 1102 did not become idle even in the absence of user activity. The effect of the bug is permanent unless a GPS location update is explicitly removed or the application is killed. The reported event traces were:

waypointsButton → waypoint_details → MenuButton → button1 (6)

waypointsButton → waypoint_details → MenuButton → BackBtn (7)

The reported bug indicated that the location updates (GPS updates) were not removed before the application became inactive. By observing the similarity between the two traces (i.e. the event sequence waypointsButton → waypoint_details → MenuButton),

it can be deduced that the bug was triggered upon arriving at a certain GUI state. After executing the event trace waypointsButton - waypoint_details - MenuButton and thereafter suspending the application by pressing the Home button, location updates are no longer needed by the application and they can be switched off. Upon inspecting the source code, it can be observed that the application had a missing code fragment for removing location updates when exiting. Thus, by using the software productivity tool in accordance with the present embodiment, the energy bug issue can be corrected by adding the release code at an appropriate location. The energy consumption graph after fixing the energy bug issue is shown in the graph 1410 of FIG. 14B. As shown in the graph 1410, the energy consumption in the POST stage 1408 is similar to the energy consumption in the PRE stage 1402, evidencing that the energy bug issue has been resolved.

[0082] The second case study involves an application Montreal Transit™ wherein five event traces with hotspots were reported from operation of the software productivity tool in accordance with the present embodiment. The energy consumption trace for one such event trace is shown in a graph 1500 in FIG. 15A. Immediately after the execution enters the REC stage 1506, potentially high E/U ratios in a period of around five seconds can be observed. 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 an abnormally high amount of energy causing the hotspot to appear during the same period. It was observed that all five reported event traces exhibited similar hotspots. On a closer inspection, it was found that the GPS location updates continue to run for a few seconds even after the application exits.

[0083] 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. Advantageously, the software productivity tool in accordance with the present embodiment 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 where the code to load advertisement was 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 reported hotspots can be best explained by such delay. When the code related to the loading of advertisements was moved to a separate asynchronous thread, the event traces which earlier exhibit hotspots no longer do so as shown in a graph 1510 of FIG. 15B.

[0084] Since I/O components and power management utilities appear to contribute significantly to the energy consumption of a mobile device, the number of system calls invoked by an event trace is used in accordance with the present embodiment as one of the guiding parameters. As a result, the test suite generated in accordance with the present embodiment should cover as many system 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, the efficacy of system call based coverage with respect to code coverage of the present embodiment was evaluated to obtain a minimal test suite for uncovering energy bugs or energy hotspots. The code coverage metric chosen was line of code (LoC) and

*EMMA*TM, a Java code coverage tool, was used to obtain LoC covered by a test suite generated in accordance with the present embodiment as compared to the total LoC of the application. It was observed that the generated test suites had a system call coverage of more than 83%, while having code coverage ranging from 11% -- 90% (see second and third columns of a table 1600 of FIG. 16). Subsequently, by manually generated additional test cases for the applications in the table 1600, it was 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), 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.

[0085] Thus, it can be seen that the present embodiment provides a software productivity tool which can quickly uncover potential energy hotspots/bugs in real-life applications. In addition, a system is also provided which advantageously quickly identifies potential energy hotspots/bugs in real-life applications for mobile devices. While exemplary embodiments have been presented in the foregoing detailed description of the invention, it should be appreciated that a vast number of variations exist.

[0086] It should further be appreciated that the exemplary embodiments are only examples, and are not intended to limit the scope, applicability, operation, or configuration of the invention in any way. Rather, the foregoing detailed description

will provide those skilled in the art with a convenient road map for implementing an exemplary embodiment of the invention, it being understood that various changes may be made in the function and arrangement of elements and method of operation described in an exemplary embodiment without departing from the scope of the invention as set forth in the appended claims.

CLAIMS

1. A software productivity tool comprising:
a detection module adapted to automatically detect a power inefficiency in a device comprising hardware components and to calculate an E/U ratio indicating power consumption to utilization of hardware components over a time period, the power consumption and the utilization of the hardware components being caused by a software program, the E/U ratio used for detection of the power inefficiency in a device on which the software program is run; and
a guidance module adapted to automatically generate at least some of a plurality of test inputs receivable by the software program and determine which of the generated test inputs will cause the power inefficiency in the device on which the software program is run.
2. The software productivity tool in accordance with Claim 1 wherein the guidance module is further adapted to extract features from the at least some of the plurality of test inputs so as to determine which of the generated test inputs will cause a power inefficiency in the device.
3. The software productivity tool in accordance with Claim 2 wherein the extracted features are used to generate a model, the model being adapted to detect test inputs which cause power inefficiencies in the software program.
4. The software productivity tool in accordance with Claim 1 wherein the guidance module ranks determines which of the generated test inputs will cause a power inefficiency in the device in response to event sequences resulting from the software program running the generated test inputs, and wherein the guidance module ranks the event sequences in response to one or more of number of system calls in the event sequence, similarity to previously identified power inefficient event sequences, and starvation of the event sequence due to unexplored system calls in the event sequence.
5. The software productivity tool in accordance with any of the proceeding claims further comprising a detection module adapted to detect a presence of at least one event sequence based on the generated model, the at least one event sequence being triggered by at least one of the plurality of test inputs and likely to cause a power inefficiency in the device.

6. The software productivity tool in accordance with any of the preceding claims wherein the software productivity tool is adapted for use on an Android platform.
7. A software productivity tool comprising:
 - a detection module adapted to automatically detect a power inefficiency in a device comprising hardware components and to detect a presence of at least one event sequence likely to cause a power inefficiency in the device on which a software program is run, the at least one event sequence being triggered by at least one input receivable by the software program; and
 - a guidance module adapted to automatically generate some of a plurality of test inputs receivable by the software program and determine which of the generated test inputs cause a power inefficiency in the device, the plurality of test inputs comprising the at least one input adapted to trigger the at least one event sequence.
8. The software productivity tool in accordance with Claim 7 wherein the presence of the at least one event sequence is detected based on the determination of the generated test inputs.
9. The software productivity tool in accordance with Claim 7 or 8 wherein the detection module is further adapted to calculate an E/U ratio used for detecting power consumption to utilization of hardware components over a time period, the device comprising the hardware components and the power consumption and the utilization of the hardware components being caused by the software program.
10. The software productivity tool in accordance with Claim 9 wherein the E/U ratio is calculated during a period before executing the software program, a period during executing the software program, and a period after executing the software program.
11. The software productivity tool in accordance with Claim 9 wherein the detection module is further adapted to detect an energy bug by comparing the E/U ratio calculated in the period before executing the software program to the E/U ratio calculated in the period after executing the software program using a statistical method, the energy bug being the at least one event sequence during which executing the software program prevents the device from being idle.

12. The software productivity tool in accordance with Claim 9 wherein the detection module is further adapted to detect an energy hotspot by analyzing the E/U ratio calculated during the period of executing the software program using an anomaly detection technique, the energy hotspot being the at least one event sequence during which executing the software program causes the device to consume power higher than required.

13. The software productivity tool in accordance with any of Claims 9 to 12 wherein the detection module is further adapted to receive a reading from a power meter attached to the device to determine the power consumption to calculate the E/U ratio.

14. The software productivity tool in accordance with any of Claims 9 to 13 wherein the software productivity tool is adapted for use on an Android platform and wherein the detection module is further adapted to receive a reading from an Android debug bridge attached to the device to determine the utilization of the hardware components of the device to calculate the E/U ratio.

15. The software productivity tool in accordance with Claim 1 wherein the E/U ratio is calculated during a period before executing the software program, a period during executing the software program, and a period after executing the software program.

16. The software productivity tool in accordance with Claim 15 wherein the detection module is further adapted to detect an energy bug by comparing the E/U ratio calculated in the period before executing the software program to the E/U ratio calculated in the period after executing the software program using a statistical method, the energy bug being the at least one event sequence during which executing the software program prevents the device from being idle.

17. The software productivity tool in accordance with Claim 15 wherein the detection module is further adapted to detect an energy hotspot by analyzing the E/U ratio calculated during the period of executing the software program using an anomaly detection technique, the energy hotspot being the at least one event sequence during which executing the software program causes the device to consume power higher than required.

18. The software productivity tool in accordance with any of Claims 1, 15 to 17 wherein the software productivity tool is adapted for use on an Android platform, and wherein the device is attached to a power meter and an Android debug bridge, and wherein the detection module is further adapted to receive a reading from the power meter to determine the power consumption to calculate the E/U ratio and receive a reading from the Android debug bridge to determine the utilization of the hardware components of the device to calculate the E/U ratio.

19. A system for identification of power inefficiencies, the system comprising:
a portable device having an energy storage device for supplying power thereto, the portable device being configured to run software programs thereon; and
a software productivity tool coupled to the portable device for identifying power inefficiencies in operation of one or more of the software programs by the portable device, the software productivity tool comprising:

a detection module adapted to automatically detect a power inefficiency in a device comprising hardware components and to detect a presence of an event sequence likely to cause a power inefficiency in the portable device on which the one or more of the software programs are run, the event sequence being triggered by an input receivable by the one or more of the software programs; and

a guidance module adapted to automatically generate some of a plurality of test inputs receivable by the one or more of the software programs and determine which of the generated test inputs are likely to cause the power inefficiencies in the portable device, the plurality of test inputs comprising the at least one input adapted to trigger the event sequence.

20. The system in accordance with Claim 19 wherein the portable device operates on an Android platform.

21. The system in accordance with Claim 20 further comprising:

a power meter coupled to the portable device for obtaining energy consumption data therefrom; and

an Android debug bridge coupled to the portable device for obtaining hardware component utilization data therefrom corresponding to utilization of hardware components of the portable device,

wherein the detection module is coupled to the power meter and the Android debug bridge for calculating an E/U ratio used for detecting power consumption to utilization of hardware components by the software program over a time period in response to the energy consumption data and the hardware component utilization data.

22. The system in accordance with Claim 21 wherein the E/U ratio is calculated during a period before executing the software program, a period during executing the software program, and a period after executing the software program, and wherein the detection module is further adapted to determine whether the event sequence is an energy bug which prevents the portable device from being idle and/or an energy hotspot which causes the portable device to consume power higher than required, the detection module detecting the energy bug by comparing the E/U ratio calculated in the period before executing the software program to the E/U ratio calculated in the period after executing the software program using a statistical method and detecting the energy hotspot by analyzing the E/U ratio calculated during the period of executing the software program using an anomaly detection technique.

23. The system in accordance with Claim 22 wherein the guidance module ranks the event sequence in response to one or more of a number of system calls in the event sequence, a similarity of the event sequence to previously identified energy bug and/or energy hotspot event sequences, and starvation of the event sequence due to unexplored system calls.