Detecting Energy Bugs and Hotspots in Control Software using Model Checking

Pascal van Gastel  
Avans University of Applied Sciences  
the Netherlands  
ppth.vangastel@avans.nl

Bernard van Gastel  
Open University  
the Netherlands  
bvg@ou.nl

Marko van Eekelen  
Open University  
the Netherlands  
Marko.vanEekelen@ou.nl

ABSTRACT

We explore a way to find energy deficiencies in code by model checking, both properties related to utilisation (energy hotspots) and not related to utilisation (energy bugs). Temporal logic properties (expressed using LTL) finds these deficiencies during the development of a system, employing a novel energy-inefficiency metric. The formal model is deduced from hardware specifications, a definition of utilisation and a control software application written in C. Model checking results in a set of traces (formally a counter-example) that can be related to the matching execution path in the source code by the software developer. The traces include energy footprints and enable developers to improve the energy efficiency of the application. A smart light system serves as a case study to evaluate the proposed approach.

CCS CONCEPTS
• Hardware → Platform power issues; • Software and its engineering → Formal software verification; Operational analysis;

KEYWORDS
energy consumption analysis, model checking

1 INTRODUCTION

Energy is an important area of concern in our world from various points of view. One view is the view of energy consumption. In our daily lives, we use dozens of devices powered by electricity. Originally, the amount of energy consumed was determined by the hardware of these devices. Although the hardware is the consumer, it is the software that has taken over the controlling function over the course of time. Becoming more and more complex, software is prone to errors, possibly leading to energy inefficient behaviour of these devices. Therefore software errors leading to abnormal energy consumption needs to be taken into account when optimising energy consumption. Although the role of software in energy reduction has become more obvious, it has not gotten much attention in the daily work of software developers yet. One of the reasons may be the lack of appropriate tooling and techniques at a mature level.

Typical energy inefficiencies for systems using components like a GPS or WiFi module are energy bugs and energy hotspots. An energy bug is a scenario in which the components of the system keep operating in a high energy state while the controlling software application has exited. An energy hotspot is a scenario in which a component temporarily is in an unnecessarily high energy state.

This paper introduces a workflow with static analysis to detect energy bugs and hotspots. An alternative metric for energy inefficiency is proposed, called $\eta$, which takes into account whether utilisation of a hardware component was on purpose, i.e. contributing to reach a certain goal. The static analysis for this workflow is automated by model checking with Spin $[6]$. Formulæ have been defined to find execution paths in which the development of $\eta$ marks energy bugs or hotspots. The proposed way of working is not limited by the number of test cases defined. Nor is it by a complex test set-up since detection does not require hardware presence.

The output of the model checker consists of energy consumption, utilisation and $\eta$ traces for root cause analysis, and can serve as test cases for the system under test. A case study has been conducted to validate the workflow on a control software application, which is, in general, less complicated than the development of other I/O-intensive software like smartphone applications.

The contributions of this paper are a practicable and goal related energy inefficiency metric and a static analysis based method for energy bug and hotspot detection. The applied static analysis provides output that defines test cases for energy related testing.

We will start with describing the scope of the detection of energy bugs and hotspots. With well defined targets to detect, we continue with an elaboration on the detection method. The energy inefficiency metric $\eta$ plays a key role herein and is defined in section 3. As required for automated detection by model checking, the method is then formalised. To demonstrate the proposed workflow, a description of a case study using the SPIN model checking follows in which we analyse energy inefficient implementations of a smart
light system to find the deliberately injected energy bug or energy hotspot.

2 ENERGY BUGS AND HOTSPOTS

Recently, more research attention has focussed on the role of software and software development in energy consumption. Greening of IT aims to find possibilities for optimising energy consumption of software-intensive systems. This section introduces two energy deficiencies for which a detection workflow is proposed later in this paper: energy bugs and energy hotspots.

For software development, code-level guidelines for energy reduction exist, like cleaning up useless code and data, monitoring appropriate resources and reduction of data being transferred [1]. This approach, however, may have limited impact on energy usage according to the authors, especially when scaling up to larger sized and more complex software systems. Therefore, power consumption related to the invocation of I/O resources should be taken into account. Although, the CPU would be the first component to consider at first sight, there is a huge amount of systems consisting of peripherals with relatively high power consumption. For instance: embedded systems, like smart lighting, mobile systems such as smartphones and tablets with components like a GPS or WiFi module [1].

Regarding a resource-usage based model, the basis for this paper are energy bugs and energy hotspots as proposed by Banerjee et al. [2]:

Energy bug a scenario with persistently high energy inefficiency, even after the application has completed execution.

Energy hotspot a scenario with a temporary high level of energy inefficiency within the execution time of the program.

A proposal to detect energy bugs and hotspot consists of automatic generation and execution of test cases [2]. Dedicated tooling, based on Dynodroid, systematically generates event traces that reflect I/O intensive usage of the Android app. While executing the test cases on a standard smartphone, a power meter records the power consumption. Subsequently, the resulting power-consumption trace is subject to analysis, where anomalies might indicate an energy bug or hotspot. Although the mentioned efforts of Banerjee et al. relate to mobile app development, the concepts of energy bugs and hotspots apply to any type of system that includes hardware components controlled by software. In practice, this detection method may lack viability since using power meters on real hardware devices requires a set-up that may be difficult to realise for software developers. Besides, dynamic analysis using profilers can only be applied on (partly) completed applications and is limited to available test cases. To overcome these limitations, we propose static analysis to detect energy bugs and hotspots in I/O intensive applications.

3 DETECTION METHOD

Using static analysis to find energy bugs and hotspots, requires energy-related information to uncover these. This section introduces an energy-inefficiency metric, whose development over time might indicate the presence of energy bugs or hotspots.

Following Banerjee et al. we focus on finding anomalies by studying the (change of) ratio between energy consumption and utilisation of the hardware component(s) over time. The utilisation part of this metric is defined in [2, p. 591] as:

\[ \text{Utilisation} (U) = \frac{\text{Weighted sum of utilisation rates of all major power consuming hardware components in a device, at a given time } t}{\text{Load}_x} \]

The weight in the utilisation rate based on the power profile of the hardware component. The utilisation rate is set to the component’s load:

\[ \text{Load}_x \text{ represents the average amount of computational work performed by the hardware component } x \text{ at a given time } t. \]

Furthermore:

\[ \text{Energy consumption to utilisation ratio, called the } E/U \text{ ratio, defined as } E(t)/U(t). \]

There are two drawbacks in the proposed E/U metric. Firstly, in case the load of a hardware component is zero, its utilisation U is zero too, which causes a division by zero in the E/U ratio. Secondly, the definition of utilisation U is defined as a weighted value of a component’s load. This lacks indication whether this load was on purpose, i.e. contributing to reach the goal.

To cope with the mentioned drawbacks of the E/U ratio, this paper proposes an alternative metric to indicate energy-inefficiency. While it is common practice to use \( \eta \) as symbol for (energy-)efficiency, its counterpart (energy-)inefficiency is indicated by \( \bar{\eta} \) from here onwards. Additionally, it is proposed to introduce utilisation \( U \) as an abstract, system-independent metric for a generic definition of \( \bar{\eta} \). For energy analysis of a concrete system, utilisation gets a corresponding domain-specific implementation. As utilisation is to be used in the energy-inefficiency metric, it is expressed as a numeric value. Energy-inefficiency is defined as follows:

Definition 1 Energy-inefficiency \( \bar{\eta} \) is the degree of energy consumption not contributing to utilisation \( U \) (i.e. to reach some system goal) relative to the maximum energy consumption budget \( E_{\text{max}} \). As a formula:

\[ \bar{\eta}(t) = (1 - U(t)) \cdot \frac{E(t)}{E_{\text{max}}} \]

When energy is being consumed, a high utilisation value \( U \) results in a lower \( \bar{\eta} \) value, reflecting a situation of efficiency, while lower \( U \) values indicate a high level of energy-inefficiency. In case utilisation is maximum, i.e. 100%, \( \bar{\eta} \) is always 0. We assume that energy consumption \( E \) will never exactly be zero since any part of a control system powered by electricity consumes some power by definition, even in case the system is in an idle or stand-by state.

To design the testing environment for energy bug/hotspot detection, an agent model [13] is used. Within the system border, agents play roles in satisfying the system’s goals and these can be software components or systems, devices, or human beings. The model includes a testing environment agent responsible for the domain-specific implementation of utilisation \( U \). Three types of agents are distinguished:

1. Environmental agents representing devices (e.g. actuators or sensors). The monitoring agents provide input for utilisation.
The ones with a controlling responsibility invoke functions on hardware components and influence energy consumption accordingly.

(2) An environmental agent representing the degree of the system’s utilisation \( U \), from this point onwards referred to as the utilisation agent. This agent is responsible for the domain-specific implementation related to the system under test. For example, in the case of driving a car to a destination, utilisation is calculated on the physical speed of the car relative to a maximum speed. An environmental agent is responsible to provide the currently measured speed. More advanced, utilisation may be defined as the physical speed in combination with being on track on a calculated optimal route. The utilisation agent combines the measured speed (by an environmental agent) with an route deviation indication (calculated by some software agent). To minimise dependence on the system, the utilisation agent should include as little knowledge about measurement details as possible. This increases the generality of algorithm to detect energy bugs and hotspots. It should only calculate utilisation \( U \) based on the weighted ratio between measured values and their corresponding maxima.

(3) The software application as a (set of) software agents, with capabilities to invoke functions on controllable agents based on information from monitoring agents.

The utilisation agent is not part of the system under test since it only has a responsibility in the proposed environment for energy bug/hotspot detection. Note that in this context testing only includes those tests needed to detect energy bugs and hotspots. The software is assumed to be correct and free of errors regarding functional requirements.

High values in the evolution of \( \eta \) might indicate the presence of energy bugs of hotspots. When a value is considered ‘high’ depends on the maximum allowed inefficiency level that represents the border between efficiency and inefficiency, denoted as \( \eta_{\text{lim}} \). It is set to 100% for this paper, i.e. focus is only at situations where energy is being consumed without any utilisation. Lower values could be used to find scenarios for optimisation, due to a relatively high amount of energy is needed for some utilisation level. Choosing an appropriate value for \( \eta_{\text{lim}} \) may depend on various factors, like the domain or availability of engineering resources. A trace reflecting an energy bug shows \( \eta \) increasing to/above the inefficiency level \( \eta_{\text{lim}} \) and remaining at/above it, even after application exit. The trace for an energy hotspot is different in the sense that \( \eta \), after reaching/exceeding the inefficiency level, decreases at some point in time to a value underneath it before the program ends. A software application in which \( \eta \) never reaches/exceeds the inefficiency level is said to be free of energy bugs and hotspots. Fictitious \( \eta \)-traces in figures 1a, 1b and 1c visualise the three trace types.

Next to the inefficiency level, the time dimension requires additional attention for hotspot detection. Two peaks reaching the inefficiency level \( \eta_{\text{lim}} \) are included in figure 1b. But, are both true hotspots? There are two ways in which energy can be consumed by a hardware component [4]. Firstly, invoking a function on that directly induces incidental (or time-independent) energy consumption.

A constant amount of energy is consumed with the function invocation. Secondly, time-dependent energy consumption: the hardware component is in a state in which it consumes a certain amount of energy per time unit. A trace for a system exposing incidental energy consumption may show a short peak like the first one in figure 1b, but it may not be desirable to mark this as an energy hotspot. It could be a case of incidental energy consumption. The second peak on the other hand, being at the inefficiency level for a longer time period, could be a true hotspot. A hotspot is detected when the system is at (or above) the inefficiency level for a time span of minimum length (\( t_{\text{min}} \)).

4 FORMALISING THE DETECTION

Energy bug/hotspot detection using static analysis requires appropriate tooling, since the process is too complex and error prone to be executed manually. When tooling is to be used, a corresponding formalisation of the detection method is needed. This section describes the choice for model checking and presents a formalisation of the detection method of the previous section.

To model energy consumption various approaches exist [7]. We use an event-based model. Energy consumption is modelled as a finite state machine (FSM): a set of states each reflecting a level of energy consumption, in relation to events that cause state transitions, for instance system calls to hardware components.

Adding software application models to the models for energy consuming hardware components in an event-based power model facilitates the searching of energy bugs and hotspots for all execution paths of the software under test. As mentioned, we apply event-based power modelling, in combination with model checking on \( \eta \)-evolution properties to find paths revealing energy bugs/hotspots.

The formalisation of the agents and definition of properties to detect energy bugs and hotspots is based on the work of Espada et al. [3]. An FSM \( M \) for an agent is a tuple \( M = (\Sigma, L, \rightarrow, s_0) \) where
- \( \Sigma \) is a set of states,
- \( L \) is a set of transition labels,
- \( \rightarrow \subseteq \Sigma \times L \times \Sigma \) is the transition relation, and
- \( s_0 \in \Sigma \) is the initial state.

The operational semantics of an FSM \( M \) is defined as the set of all possible traces (paths), either finite or infinite, that can be constructed by iteratively applying the transition relation from the initial state: \( O(M) = \{ \pi | s_0 \rightarrow s_1 \rightarrow ... \} \). Traces match execution paths in the software application. A map \( \pi : \mathbb{N} \rightarrow \Sigma \) associates natural numbers to the corresponding states in the trace, i.e. \( \pi(i) = s_i \). Transition labels are not needed here, so they are discarded from here onwards. Detection of energy bugs an hotspots
targets at \( \bar{\eta} \)-evolution over time, based on corresponding evolutions of \( E \) and \( U \). To incorporate this in the formalisation, let \( E(t) \in \mathbb{R}_{>0} \) be the level of energy consumption \( E \) of hardware component \( c \) at time \( t \) and \( U(t) \in \mathbb{R}_{>0} \) be the utilisation level \( U \) at time \( t \). In the system under test, a set of hardware components \( C \) may be involved, each with a specific \( E_c(t) \). The total energy consumption is defined as \( E(t) = \sum_{c \in C} E_c(t) \). We assume that hardware components always have some energy consumption, \( E(t) > 0 \).

In itself, model checking of properties on FSMs abstracts from the notion of time by reasoning about sequentiality of states. Because detecting energy bugs and hotspots involve \( E(t) \) and \( U(t) \), an explicit notion of time is required here. The trace \( \pi \) needs to be linked, in two steps, both with \( E(t) \) and \( U(t) \).

Firstly, a trace \( \pi \) can be executed at any moment in time and to associate an execution of \( \pi \) with concrete time, each state in \( \pi \) is related to a time point. Formally (from [3, p. 136]):

**Definition 4**

Given an execution function \( e(\pi, i) \), an execution \( e \) of this trace \( \pi \) is given by a function \( e: \{ \pi \}, i \rightarrow \mathbb{R} \) that associates each state \( \pi(i) \) with a time instance being the sum of all time spans between all states preceding \( \pi(i) \), with a time span value in \( \mathbb{R}_{>0} \). Note that the calculated time instances strictly increase and \( e(\pi, 0) \) is defined as 0. It is assumed that \( e(\pi, i) \) represents the creation time of \( s_i \) and \( t_i \) is a short notation for that time instant.

Secondly, the addition of time to the execution of a trace by \( e(\pi, i) \) introduces the possibility to relate executions to \( E(t) \) and \( U(t) \) by associating each point in time of \( e(\pi, i) \) with the values of \( E \) and \( U \) at that same instant. The energy-inefficiency \( \bar{\eta} \) at a certain point in time is calculated from \( E(t) \) and \( U(t) \).

**Definition 3**

Given an execution function \( e(\pi, i) \), a state \( s_j \) is associated with the energy consumption \( E \) at the time instant for that state, i.e. \( E(t) = e(\pi, i) \).

**Definition 4**

Given an execution function \( e(\pi, i) \), a state \( s_j \) is associated with the utilisation \( U \) at the time instant for that state, i.e. \( U(t) = U(e(\pi, i)) \).

**Definition 5**

Given a trace \( \pi \) and let \( \bar{\eta}_{lim} \) be the efficiency/inefficiency border, an energy bug occurs in \( \pi \) when

1. \( \exists x (\pi(x) \neq (\bar{\eta}(x) < \bar{\eta}_{lim})) \), and
2. \( \exists y < x (\pi(y) \neq (\bar{\eta}(y) < \bar{\eta}_{lim})) \)

**Definition 6**

Given a trace \( \pi \), let \( \bar{\eta}_{lim} \) be the efficiency/inefficiency border and \( t_q \) is the minimum length for the time span of an energy hotspot, an energy hotspot occurs in \( \pi \) when

1. \( \exists x (\pi(x) = (\bar{\eta}(x) < \bar{\eta}_{lim})) \), and
2. \( \exists \exists y < x (\pi(y) = (\bar{\eta}(y) < \bar{\eta}_{lim})) \), and
3. \( \forall x < y \in \pi (\bar{\eta}(z) = (\bar{\eta}(x) < \bar{\eta}_{lim})) \), and
4. \( (y - x) \geq t_q \)

The LTL -property to detect energy bugs using definition 5 has been defined as \( [\text{end}_{of}_{program} \rightarrow G (\bar{\eta} < \bar{\eta}_{lim})] \). The end_of_program proposition has been added to avoid detection of false negative scenarios in which the controlling application ends with a high energy inefficiency level and detection ends due to a time limit. The LTL -property to detect energy hotspots requires a function, counting the number of energy hotspots on trace \( \pi \) and taking \( t_q \) into account. It has been defined as \( [\text{nr}_{hotpt}(\pi) = 0] \).

### 5 DETECTION WORKFLOW WITH CASE STUDY

Evaluation of the proposed method for energy bug and hotspot detection aims at testing that model checking the defined LTL-formulae on \( \bar{\eta} \)-development reveals true energy bugs/hotspots and that its output provides pointers to find causes in source code. A simple control software system is the system under test. A set of energy inefficient implementations has been developed; each version in the set was deliberately injected with either one energy bug or one energy hotspot. The workflow has been applied to each implementation to find the injected energy bug/hotspot. After fixing the inefficiency, the test case was rerun to check for improved energy consumption. This section describes one test case for an energy bug and one for a hotspot.

The control software system is a smart light system with a motion detector that switches on a light in case somebody has been detected (the active state). It implements a time-boxed delay state of 2 time units when light is on and no person is being detected anymore. This state is an example of incidental energy consumption with accepted high energy inefficiency. In case after the delay somebody is detected again, the system returns to the active again. Otherwise, the light is turned off (idle state). The motion detector is simulated to facilitate testing; this does not limit the experiment. The light system is the PHILIPS HUE. The system software has been implemented in the C programming language.

A summary of the smart light system PROMELA-model and implementation artefacts is provided by table 1. The motion detector agent and light system agent have been modelled as inline definitions since limiting the number of processes enhances performance of the Spin-model checker.

```
inline light_switch(new_light_state) {
  if
    : new_light_state == LIGHT_ON ->
    energy_E = max_E_light;
    : new_light_state == LIGHT_OFF ->
    energy_E = min_E_light;
    fi;
    ...
  }
```

The model defines variables for \( E, U \) and \( \bar{\eta} \) which are written to output trace each time unit starting at \( t_0 \) to enable (graphical) analysis. The model artefact for the motion detector non-deterministically generates values indicating a person’s presence/absence.

### Table 1: Smart light implementation and model artefacts.

<table>
<thead>
<tr>
<th>Component</th>
<th>Agent</th>
<th>Smart light implementation</th>
<th>Promela model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Type</td>
<td>Artifact</td>
<td>Responsibility</td>
</tr>
<tr>
<td>motion detector</td>
<td>sensing environmental agent</td>
<td>component</td>
<td>indicates a presence/absence, based on values of the model checking output trace</td>
</tr>
<tr>
<td>Thsys</td>
<td>controlling environmental agent</td>
<td>component</td>
<td>linked to the real Thsys light system</td>
</tr>
<tr>
<td>controller</td>
<td>software agent</td>
<td>component</td>
<td>sends input from motion detector and switches light on/off accordingly</td>
</tr>
<tr>
<td>utilisation</td>
<td>utilisation agent</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>
Detecting Energy Bugs and Hotspots in Control Software...

```c
inline motdet_SomebodyThere() {
  if :: skip -> somebodyThere = 0;
  if :: skip -> somebodyThere = 1;
  fi;
  utilisation(somebodyThere); // update utilisation
}
```

These values are written to the output trace as well serving as a test case for the smart light system implementation. Its motion detector component is able to read the trace and act accordingly. Each time a new presence/absence value is generated by the model artefact for the motion detector, it invokes the utilisation definition in the PROMELA model to update $U$. The following PROMELA-code fragment shows a part of the inline definition of utilisation:

```c
inline utilisation(u_value) {
  utilisation_U = (100*u_value)/max_u_input_value;
  TRACE_U_UPDATE // also traces time instance
  ...
}
```

In this case, only one value is input to utilisation $U$, i.e. somebody’s absence/presence provided by the motion detector artefact. In case utilisation has a more complex definition, all artefacts providing input invoke the single utilisation artefact explicitly to recalculate $U$.

To evaluate the proposed energy bug/hotspot detection method, an energy bug and hotspot were injected into implementations corresponding PROMELA models of the controller component. Model checking resulted in error trails, and the corresponding execution traces lead to the manifestation of the energy bug and hotspot. To judge validity of the finding, the related motion detector tracing based workflow presented is an effective way to detect energy bugs and hotspots. The case study showed that a model-based energy consumption analysis method can encounter this disadvantage [10]. In earlier research [9], Nakajima proposed a power consumption automation and a variant of linear temporal logic, Weighted Linear Temporal Logic with freeze quantifiers (FLWTL). This logic supports constraints on variables in the definition of properties via freeze quantifiers. A more recent study [10] explores the possibilities and restrictions of model checking FLWTL formulas using the Real-Time Maude tool.

Another alternative to runtime profilers is a type system designed to precisely predict energy consumption of software by Van Gastel et al. [4]. The type system is hardware-parametric. Using a dependent type system, one can define reusable energy signatures for functions. This approach is modular and composable. The semantics includes an explicit construct for invoking hardware components [12]. In this approach [5], these hardware components are modelled as finite state machines. A similar approach is used in [8] to analyse memory consumption of programs.

6 RELATED WORK

Software profiling is a way to identify possibilities to optimise energy-consumption. EPROF is an energy profiler for smartphone apps aiming to determine where in the app energy is spent [11]. In-depth research using EPROF revealed that major part of the energy consumption of smartphone apps relates to I/O components like WiFi and GPS [11].

Model Based Energy Testing (MBET) is a way to verify energy consumption by using behaviour models or the software under test itself for test case generation and execution [14]. An implementation of this concept is a tool chain that checks whether actual energy consumption is within expected limits [3]. Based on a model of the application, Spin [6] automatically generates test cases representing user actions. Subsequently, the test cases are being executed and timestamped traces of states of the device are recorded. Additionally, a power meter measures the energy consumption during the test case execution. Combining these energy measurements with the timed traces results in an enriched trace to validate against a given set of energy properties.

When using a runtime profiler, the coverage of energy verification is limited to the supplied test cases. A model-based energy consumption analysis method can encounter this disadvantage [10]. In earlier research [9], Nakajima proposed a power consumption automation and a variant of linear temporal logic, Weighted Linear Temporal Logic with freeze quantifiers (FLWTL). This logic supports constraints on variables in the definition of properties via freeze quantifiers. A more recent study [10] explores the possibilities and restrictions of model checking FLWTL formulas using the Real-Time Maude tool.

7 CONTRIBUTION

The contribution of this paper to improve energy-aware software engineering is twofold. Firstly, the proposed definition of energy inefficiency $\eta$ is solid and semantically makes sense by taking intended use of an energy consuming component into account. Secondly, the model checking based workflow presented is an effective way to detect energy bugs and hotspots. The case study showed that the presence of true energy bugs/hotspots can be detected, with a key role for the energy inefficiency metric $\eta$. The detection method is technology agnostic, for instance with respect to hardware, platform, programming language, and does not require a complex test set-up.
REFERENCES


