Automatic Energy-Hotspot Detection and Elimination in Real-Time Deeply Embedded Systems

Mohsen Shekarisaz
School of Electrical and Computer Engineering, University of Tehran
Tehran, Iran
Email: shekarisaz@ut.ac.ir

Lothar Thiele
Computer Engineering and Networks Laboratory, ETH Zürich
Zürich, Switzerland
Email: thiele@ethz.ch

Mehdi Kargahi
School of Electrical and Computer Engineering, University of Tehran
Tehran, Iran
Email: kargahi@ut.ac.ir

2021 IEEE Real-Time Systems Symposium (RTSS) | 978-1-6654-2802-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/RTSS52674.2021.00020

***Abstract*—Today's deeply embedded systems, with real-time interactions to the environment, are largely battery-operated, and peripheral modules like LTE, WiFi, and GPS are among the most energy-hungry components of them. These components are often under the direct control of an embedded software. Some pieces of the software program are called *energy hotspots* if they can be transformed towards better system energy consumption while leaving it logically- and temporally-correct. This paper focuses on three such energy hotspots from the peripheral module perspective. The root causes of the hotspots in the software program are misplaced driver calls: Early acquiring or late releasing of the module causes it to waste energy in the active state, having unnecessary distance between the use operations causes extra tail energy overhead, and unaccounted releasing and re-acquiring of the module causes more energy consumption in comparison to leaving the module unreleased. We provide static analysis methods for the detection and elimination of such energy hotspots automatically with regard to some relations between temporal requirements of the real-time embedded software, the time and energy specifications of the module, and the extreme (worst-case/best-case) execution times of specific pieces of the software program. Our simulation results show about 4.7 to 20 percent of energy reductions after elimination of the energy hotspots of the test programs using our proposed method.**

***Index Terms*—battery-operated, deeply embedded systems, energy hotspot, real-time systems, static analysis, worst-case execution-time (WCET)**

I. Introduction

Deeply embedded devices in the context of Cyber-Physical Systems (CPS) and Internet-of-Things (IoT) are designed to provide a wide variety of functionalities for various application types including health-care, monitoring, etc. Most these functionalities are interactions with the surrounding environment through peripheral modules (e.g. LTE, WiFi, GPS), and many of such modules are the most energy-hungry components of the systems. Accordingly, tight energy budget greatly constrains these applications, which is mainly due to required autonomy [1] through battery-based operation. Continuous operation of these devices despite their limited energy is thus a major design and implementation goal in addition to the well-known requirements of temporal and logical correctness.

Deeply embedded systems usually tend to come in the form of a micro-controller unit (MCU) augmented with the aforementioned energy-hungry peripheral modules [2]. To have a better insight into the power behavior of such systems, we mention a setup containing a 32-bit MCU and a network-based peripheral module, which are of the most common hardware units, contributing respectively about 61% and 49% in the embedded market [3]. Table I shows some average values of the power, time, and energy specifications of such typical configurations at different statuses, extracted from the datasheets. In many systems of this type, the peripheral modules consume considerable power compared to the MCU. Here, when the module is in the *Active* state, it is ready for use and consumes considerable power of about 64.3 mW. Comparing it to the MCU when it runs computational instructions (i.e. 129 mW), we find that the module consumes about 0.49 times the MCU power. Thus, a mistake of unnecessarily putting the module into the *Active* state consumes almost as much as half the energy of the busy MCU or almost 13.1 times the MCU power when it waits for a response from a peripheral module (i.e. 9.8 mW).

Discussing from a different perspective, it may seem a good decision to put the module into the *Sleep* state (where its features are disabled and it consumes a low power) when it is not needed. However, the energy consumption for doing so and putting it back into the *Active* state when it is needed again (i.e. the energy of each pair of *Active-to-Sleep* and *Sleep-to-Active* switches) is about 130 mJ; it imposes a relatively large energy overhead of about 2 times of when the module is working for a bounded sending of 1 KB of data, i.e. 64.6 mJ, which may invalidate the above decision. Moreover, after finishing each round of working of the module, it tends to consume some tail power for some duration until switching back to the *Active* state [4], [5], [6]. It is up to 36.34 mJ of tail energy for the module of Table I, which is a considerable overhead (about

978-1-6654-2802-6/21/$31.00 ©2021 IEEE
DOI 10.1109/RTSS52674.2021.00020

0.56 times of a round of working). It can be reduced if the module usages be closer to each other with regard to the tail duration (i.e. 163.5 ms). More formal details on these states will be presented in Section II.

The software part of a deeply embedded system which comprises computational instructions (on the MCU) and I/O statements (via *driver calls*, employing the peripheral modules) has a major role in the system energy usage behavior. Regarding that almost 35% of the embedded systems market in 2019 has belonged to deeply embedded systems that do not run on top of any operating system [3], the way that the embedded program utilizes the above-mentioned energy-hungry peripheral modules using driver calls greatly impacts the system energy consumption. Hence, energy-efficient embedded software design has been seriously considered in the past few years [7], [8], [9], [10], [11].

If a piece of program wastes energy by the way that it uses a peripheral module, while the waste could be prevented by using it in a different way, we are dealing with a *program energy defect*. Even though we may not be able to have a defect-free program from the energy perspective, we can identify some common programming mistakes and try to eliminate them towards energy efficiency of the software program.

Two major categories of energy defects do exist: energy bugs and energy hotspots [12]. An *energy bug* happens when a module continues to consume energy even after completion of the program execution. Such a scenario in a program happens when the module is acquired (*Sleep-to-Active*, in Table I), but the designer forgets to release it (*Active-to-Sleep*, in Table I), resulting in a *resource leak* [13], [14], [15]. The *energy hotspot*, however, is a less studied programming energy defect resulting in inefficient energy consumption during the program execution, similar to the aforementioned quantitative examples. Most energy hotspots cannot be easily uncovered by a program inspection, as they would require a more detailed timing analysis of the real-time software program to be revealed, namely the timing behavior of the software must be considered in addition to the energy considerations.

In this paper, we focus on energy hotspots, and discuss three types of such energy defects. The considered hotspot types are defined based on the locations of the peripheral module driver calls in the software program with regard to the extreme (worst-case/best-case) execution times of specific program pieces. We propose some static program analysis methods to automatically detect and eliminate such hotspots, taking into account the logical and timing requirements of the real-time embedded software in the interactions to the environment. We evaluate our approach through extensive simulations with a wide variety of test programs based on MiBench [16] benchmark suite, augmented with the module driver calls. Comparing the energy consumption of the original and modified test programs, we found up to 20 percent energy reductions, as illustrated through discussions. In summary, the contributions of this paper are as follows:

- Formal definition of three types of program energy hotspots in real-time deeply embedded systems;
- Presence of a static program analysis method for the detection of the hotspots;
- Provision of algorithms to automatically removing the identified hotspots and building an energy-improved program which remains logically and temporally correct; and
- Presence of extensive simulation results to demonstrate the efficacy of the proposed energy-hotspot detection and elimination approaches in reducing the system energy consumption for different test programs.

TABLE I: The power, time and energy specifications of BG96 LTE module [17] and AT91SAM7X256 32-bit MCU [18].

Component	Status	Specification		
		Power (mW)	Time (mS)	Energy (mJ)
Peripheral Module	*Sleep-to-Active* (*Acquire*)	76.95	1688.5	129.93
	Active-to-Sleep (*Release*)	8.55	9	0.076
	Working (*Use*)	296.4	218	64.6
	Tail	222.3	163.5	36.34
	Active	64.3	-	-
	Sleep	6.46	-	-
MCU	*Running*	129	-	-
	Waiting	9.8	-	-

The remainder of this paper is organized as follows. In Section II, we give our system model and problem statement. Section III provides the detailed definition of energy hotspots, and Section IV provides the algorithms for detection and elimination of the hotspots. In Section V, we illustrate the effectiveness of our algorithms through simulation results. We discuss the related works in Section VI. Finally, Section VII concludes the paper and presents some ideas for future work.

II. System Model and Problem Definition

In this section, we present the model of the system, including the characteristics of the hardware components and those of the single-threaded embedded software program. Then, we define the problem studied in this paper.

A. Platform Model

The target platform consists of a computational unit, denoted as *MCU*, and a single peripheral module[1] (resource), shown as r. We suppose six states for r: *Sleep*, with a power usage of Pow_S, in which r is not ready for use; *Active*, with a power usage of Pow_A, in which r is ready for use; *Working*, with a power usage of Pow_W, that is the only state in which r is used; *Acquire*, with a power usage of Pow_{Acq}, in which r is being switched from *Sleep* to *Active*; *Release*, with a power usage of Pow_{Rel}, in which r is being switched from *Active* to *Sleep*; and *Tail*, with a power usage of Pow_{Tail}, in which r lingers before switching back from *Working* to *Active*.

Fig. 1 illustrates the state model of r by a continuous timed automata (X represents the clock). The programmer can control or manipulate some states of r through *driver calls*, each of which is modeled as a single statement. Calling $Acquire(r)$ changes the state of r from *Sleep* to *Active*, however, after a delay of D_{Acq} and with an energy overhead of $E_{Acq} = D_{Acq}Pow_{Acq}$. Calling $Release(r)$ puts back r into *Sleep*,

[1]With no loss of generality, this study focuses on a single module, however, all the discussions will be valid for systems containing more such modules.

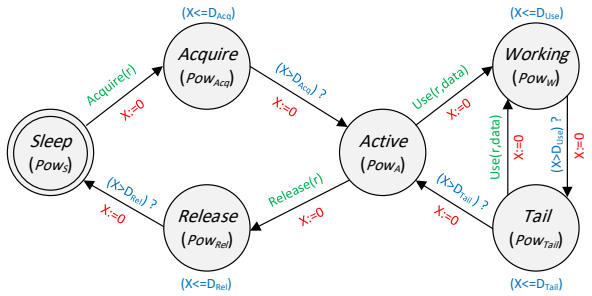

Fig. 1: State machine of module r.

however, after a delay of D_{Rel} and with an energy overhead of $E_{Rel} = D_{Rel}Pow_{Rel}$. Finally, by calling $Use(r,data)$ when r is in *Active*, r is put into *Working* to perform some operations on *data*; it takes a maximum delay of D_{Use} to perform the operation, and it consumes $E_{Use} = D_{Use}Pow_W$. After passing D_{Use}, r starts to return back to *Active*, which takes a delay of D_{Tail} to complete the transition. However, if another $Use(r,data)$ is called in between, the module does not complete the transition to *Active*, and returns to *Working*. Suppose $t = 0$ is the time when the i^{th} $Use(r,data)$ is finished. We show the time interval from $t = 0$ to the next call of $Use(r,data)$ as $Intvl_i$. Therefore, after finishing the i^{th} $Use(r,data)$, r consumes some tail energy $E_{Tail}(Intvl_i)$, modeled as:

$$E_{Tail}(Intvl_i) = \begin{cases} Pow_{Tail}|Intvl_i| & 0 \le |Intvl_i| < D_{Tail} \\ Pow_{Tail}D_{Tail} & |Intvl_i| \ge D_{Tail} \end{cases} \quad (1)$$

According to (1), if $Intvl_i$ is shorter than D_{Tail}, r uses less tail energy. We consider $Intvl_i = D_{Tail}$ for the last $Use(r,data)$ before $Release(r)$, meaning that the maximum tail energy is imposed by that.

Regarding *MCU*, it can run either computational instructions or module-related statements (i.e. $Acquire(r)$, $Use(r,data)$ or $Release(r)$), causing two *MCU* states: *Busy*, in which *MCU* executes a computational instruction and its power consumption is Pow_B; and *Idle*, in which *MCU* waits for r to respond and its power consumption is Pow_I. For every module-related statement, *MCU* spends a corresponding delay (i.e. one of D_{Acq}, D_{Use} or D_{Rel}) in the *Idle* state[2].

B. Embedded Software Model

A program P runs as the embedded software on *MCU* and uses r through performing some module-related statements as driver calls. The driver calls include $Acquire(r)$, $Release(r)$, and $Use(r,data)$; each call comprises several writes to the module hardware control registers, leading to intermediate module states, a matter which is not the focus of this study.

Depending on the program inputs, P has several distinct paths to take at each run. We denote these paths as p_k, $k = 1,2,....$ The energy consumption of the embedded system in path p_k, $Energy(p_k)$, is calculated as the sum of energy consumption of the hardware components in that path, namely:

$$Energy(P,p_k) = Energy_r(P,p_k) + Energy_{MCU}(P,p_k), \quad (2)$$

where

$$Energy_r(P,p_k) = Num_{Acq}E_{Acq} + Num_{Rel}E_{Rel} + \sum_{i=1}^{Num_{Use}} (E_{Use} + E_{Tail}(Intvl_i)) + t_S Pow_S + t_A Pow_A, \quad (3)$$

$$Energy_{MCU}(P,p_k) = t_I Pow_I + t_B Pow_B.$$

In (3), Num_{Acq}, Num_{Rel}, Num_{Use}, t_A, t_S, t_I, and t_B, respectively, are the number of $Acquire(r)$ statements, number of $Release(r)$ statements, number of $Use(r,data)$ statements, time spent by r in the *Active* state, time spent by r in the *Sleep* state, time spent by *MCU* in the *Idle* state ($t_I = Num_{Acq}D_{Acq} + Num_{Rel}D_{Rel} + Num_{Use}D_{Use}$), and time spent by *MCU* in the *Busy* state, in path p_k of P.

We represent the control flow of P using a modified control flow graph $MCFG =< N, E_{MCFG} >$, where N is the set of nodes, augmented with two symbolic *Start* and *End* nodes, and $E_{MCFG} = \{(n_i,n_j)$: there is a control flow from n_i to $n_j\}$ is a set of directed edges. Each node n_i is formatted as a tuple: $n_i =< e_i, type_i >$, in which $type_i \in \{Acq, Rel, Use, Comp, Branch, Join\}$ denotes the node type, where *Acq* nodes contain a single $Acquire(r)$ statement, *Use* nodes contain a single $Use(r,data)$ statement, *Rel* nodes contain a single $Release(r)$ statement, *Comp* nodes are basic blocks, composed of only computational statements, *Branch* nodes contain a single branch-related statement, and *Join* nodes are dummy nodes (with no statements) where different branch are joined together. Furthermore, e_i is the node execution time (or equivalently, delay, for module-related nodes). As an example, consider the program in Fig. 2a. The corresponding *MCFG* with annotated node types is shown in Fig. 2b. We need a few definitions.

Definition 1. From the module perspective, a logically correct program P does have the correct order of module-related statements, namely: Its corresponding *MCFG* represents no *resource leak* [13], i.e. in each path of *MCFG*, corresponding to an *Acq* node, there is a subsequent *Rel* node; disregarding *Comp* nodes, neither there are two consecutive *Acq* nodes, nor there are two consecutive *Rel* nodes; and the *Use* nodes are located between an *Acq* node and the subsequent *Rel* node.

Definition 2. Suppose A and B are two sets of nodes. A partial *MCFG*, shown as $\partial MCFG(A,B)$, is defined as a sub-graph of *MCFG* that contains a node or an edge of *MCFG* if and only if it is in some path between some node in A and some node in B (we use $\partial MCFG(n_i,n_j)$ when $n_i \in A$, $n_j \in B$ and $|A| = |B| = 1$).

Definition 3. Suppose a logically correct program P. Regarding the corresponding *MCFG*, $next_{Acq}(n_i)$ is the set of immediate *Acq* nodes after n_i in the paths of $\partial MCFG(n_i,End)$; in case that one or more paths in the partial *MCFG* does not contain any *Acq* node, *End* is put in the set as an

[2]In a previous study [19], the authors have considered non-blocking module-related statements, where *MCU* continues to execute computational instructions until the driver call finishes after its delay. Such an assumption adds some trivial calculations related to the *Busy* time of *MCU*, which we decided to ignore it in the current study for the sake of clearer presentation and its lack of relevance to the contribution of this paper.

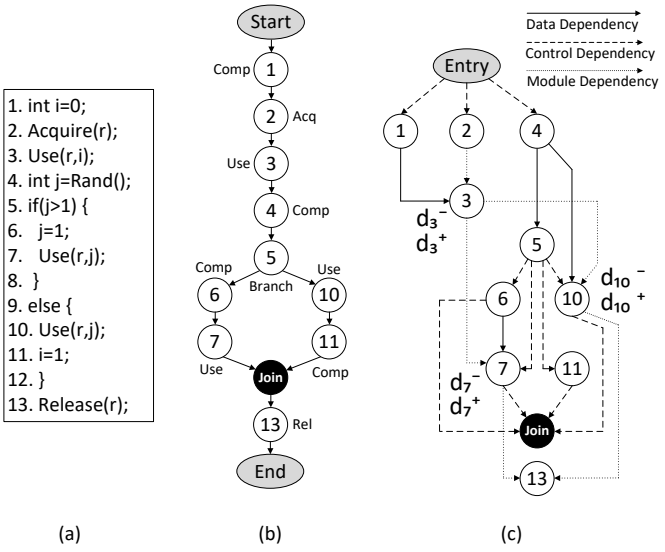

Fig. 2: The embedded software program model: (a) A sample program, (b) The annotated *MCFG*, (c) The annotated *MPDG*.

alternative. $next_{Rel}(n_i)$ is the set of immediate *Rel* nodes after n_i, with no module-related nodes in between, in the paths of $\partial MCFG(n_i, End)$. $next_{Use}(n_i)$ is defined as the set of immediate *Use* nodes after n_i, with no module-related nodes in between, in $\partial MCFG(n_i, End)$. In a similar fashion, we can define $prev_{Acq}(n_i)$ as the set of immediate *Acq* nodes before n_i, with no module-related nodes in between, in the paths of $\partial MCFG(Start, n_i)$, $prev_{Rel}(n_i)$ as the set of immediate *Rel* nodes before n_i in the paths of $\partial MCFG(Start, n_i)$ where in case that one or more paths in the partial *MCFG* does not contain any *Rel* node, *Start* is put in the set as an alternative, and $prev_{Use}(n_i)$, which is the set of immediate *Use* nodes before n_i, with no module-related nodes in between, in $\partial MCFG(Start, n_i)$.

Definition 4. Partial WCET $\partial WCET(A,B)$ is the WCET of $\partial MCFG(A,B)$, for any path between any node in A and any node in B, ignoring the execution time of nodes in A and B [20] (we use $\partial WCET(n_i,n_j)$ when $n_i \in A$, $n_j \in B$ and $|A| = |B| = 1$). Partial BCET $\partial BCET(A,B)$ is also defined in a similar fashion. If either $A = \emptyset$ or $B = \emptyset$, we suppose $\partial BCET(A,B) = \partial WCET(A,B) = 0$.

Dependencies between the statements of P are represented by a program dependence graph [21] augmented with a symbolic *Entry* node, $MPDG =< N, E_{MPDG} >$, where the set of nodes N is similar to that of *MCFG* and the set of directed edges is shown by $E_{MPDG} = \{(n_i,n_j) : n_j$ has some type of dependency to $n_i\}$ (see Fig. 2c). The dependency types include, data, control (regarding *Branch* and *Join* nodes), and module dependencies. The latter type of dependency can more formally be defined as follows. Node n_j is module dependent on node n_i, i.e. it is not allowed to replace them with each other during the program modification, if:

- $type_i, type_j \in \{Acq, Use, Rel\}$;
- $type_i$ is *Acq* and n_j is in $next_{Use}(n_i)$ or $next_{Rel}(n_i)$, $type_i$ is *Use* and n_j is in $next_{Use}(n_i)$ or $next_{Rel}(n_i)$, and $type_i$ is *Rel* and n_j is in $next_{Acq}(n_i)$.

In addition to the logical aspects mentioned above, the program P has some time constraints too. According to the extensive literature dealing with the usefulness of a program timing behavior, a correct behavior is not necessarily specified by meeting a deadline [22]. Rather, a more meaningful behavior is specified by the time/utility function (TUF) of the program, which assigns a usefulness to each response time of it. TUF is a generalization of the deadline constraint, one of the simplest forms of which is specified with an initial time and a deadline; namely, the program P has two time constraints of D^- and D^+, respectively representing the earliest and latest valid completion times of it (we reach the classic deadline by setting D^- to zero). As we will see, since an effect of our hotspot elimination method may be shortening the given program length in some paths depending on the type of eliminated energy hotspot, with the more general time constraints considered in this paper the method can be applied to a broader range of real-time applications.

More specifically, we suppose a best-case execution time (BCET) of C^- and a worst-case execution time (WCET) of C^+ for P. The execution of P is useful only if it is finished at a relative time between $[D^-, D^+]$, namely $C^- \geq D^-$ and $C^+ \leq D^+$; otherwise, P is temporally incorrect. Therefore, all modifications of P towards energy improvement should respect these temporal constraints, i.e. the modifications are restricted to only utilize the slacks $C^- - D^-$ and $D^+ - C^+$ of P in the corresponding directions.

Further, for each node n_i of type *Use*, based on the program specifications, the programmer has to respect to two predetermined time constraints of d_i^- and d_i^+, where for the earliest completion time $c_i^- = \partial BCET(Start, n_i) + e_i$ we should have $c_i^- \geq d_i^-$, and for the latest completion time $c_i^+ = \partial WCET(Start, n_i) + e_i$ we should have $c_i^+ \leq d_i^+$. In *MPDG*, each node n_i of type *Use* is annotated with the corresponding time constraints of d_i^- and d_i^+ (see Fig. 2c), which are to be respected in any modifications of P as well.

Definition 5. From the module perspective, we call a program P *temporally correct* if it has the following properties: $C^- \geq D^-$, $C^+ \leq D^+$, and for each *Use* node n_i (which performs an input/output operation) of P, we have $c_i^- \geq d_i^-$ and $c_i^+ \leq d_i^+$.

C. Problem Definition

In general, a piece of program whose energy consumption is higher than necessary, and the unnecessary energy can be reduced by manipulation of that piece of program is called an *energy hotspot*. The problem of detection and removal of energy hotspots according to the general definition (or equivalently, the problem of program energy optimization) is intractable. Therefore, this study focuses on specific hotspots related to r based on the model of this section.

The first part of our problem is thus to define energy hotspots which relate to the places of $Acquire(r)$, $Use(r, data)$

and $Release(r)$ in P, and the partial BCET/WCET of some program pieces between the module-related statements.

Afterwards, we consider a logically and temporally correct program P of a deeply embedded system which may suffer from the module-related energy hotspots and is to be modified towards eliminating them. Therefore, we should provide some methods for detection of such hotspots. We also consider some program manipulation actions restricted to the module-related statements with the consideration that the program should remain logically the same and temporally correct after the manipulations; these actions include moving the module-related nodes of *MCFG* of P forward and backward, replicating/merging the nodes, and removing some nodes of type *Acq* and *Rel*. A modified version of P, say P', remains temporally correct if $C'^{-} \geq D^{-}$, $C'^{+} \leq D^{+}$, and for any *Use* node n_i, we have $d_i^{-} \leq c_i'^{-} \leq c_i'^{+} \leq d_i^{+}$, where C'^{-}, C'^{+}, $c_i'^{-}$ and $c_i'^{+}$, are respectively, the BCET, WCET, the earliest completion time, and the latest completion time of n_i of P'.

Overall, we get a program code P, detect its module-related energy hotspots one-by-one, and employ the mentioned manipulations towards eliminating each of the detected hotspots. This way, the goal is to construct a modified program P', such that the energy consumption of P' for each individual path p_k (which is activated based on specific inputs) gets better than (or remains the same as) that of P for the same inputs, namely:

$$Energy(P', p_k) \leq Energy(P, p_k), \forall p_k \in P, P' \tag{4}$$

III. Energy Hotspots

In this section, we define three module-related energy hotspots, namely, $\text{Hotspot}_{\text{Tail}}$, $\text{Hotspot}_{\text{Sleep}}$, and $\text{Hotspot}_{\text{Active}}$.

A. $Hotspot_{Tail}$

Energy hotspot of type $\text{Hotspot}_{\text{Tail}}$ is defined based on the time distance between two subsequent $Use(r, data)$ statements in P with regard to the tail energy. Suppose the i^{th} and $(i+1)^{th}$ $Use(r, data)$ statements in a path of P. Since r consumes $E_{Tail}(Intvl_i)$ after completion of the i^{th} $Use(r, data)$ statement (with the maximum of $E_{Tail}(D_{Tail})$, see (1)), there is some energy-inefficiency between the $Use(r, data)$ statements. If we manipulate the program code by moving the i^{th} $Use(r, data)$ statement forward just beside the $(i+1)^{th}$ $Use(r, data)$ statement, then we will have $Intvl_i = 0$ and $E_{Tail}(Intvl_i) = 0$, and thus the hotspot is completely eliminated. However, this might not be feasible because of some dependencies between the computational and $Use(r, data)$ statements and the time restrictions, which are discussed in detail in Section IV.

Proposition 1. *Consider $\partial MCFG(n_i, next_{Use}(n_i))$ as a piece of P, where n_i is a Use node. Then we have a $Hotspot_{Tail}$ between n_i and $next_{Use}(n_i)$, if $\partial BCET(n_i, next_{Use}(n_i)) > 0$. The piece of program gets more energy-efficient if we move n_i forward in MCFG towards minimizing $\partial BCET(n_i, next_{Use}(n_i))$ (and equivalently, $\partial WCET(n_i, next_{Use}(n_i))$).*

Proof. If $\partial BCET(n_i, next_{Use}(n_i)) > 0$, then r certainly consumes tail energy for all paths in $\partial MCFG(n_i, next_{Use}(n_i))$, either the shortest or the others with higher execution times. By moving n_i forward (as far as dependencies and time restrictions permit), it is placed closer to the elements of $next_{Use}(n_i)$; thus, $Intvl_i$ with respect to every element of $next_{Use}(n_i)$ gets smaller, and therefore, r consumes less (or the same) tail energy in each individual path which may be executed at runtime. If n_i reaches a *Branch* node during the forwarding, then it can be replicated and put at the start of all the paths after the *Branch* node. While traversing the updated *MCFG*, one of the replicated nodes is considered as the current node n_i, and it is moved forward towards minimizing $Intvl_i$. The same procedure will be followed for the other replicated nodes. If n_i reaches a *Join* node during the forwarding, it is not moved anymore because of the control dependencies of *MPDG*. □

Proposition 2. *Consider $\partial MCFG(prev_{Use}(n_i), n_i)$ as a piece of P, where n_i is a Use node. Then we have a $Hotspot_{Tail}$ between $prev_{Use}(n_i)$ and n_i, if $\partial BCET(prev_{Use}(n_i), n_i) > 0$. The piece of program gets more energy-efficient if we move n_i backward in MCFG towards minimizing $\partial BCET(prev_{Use}(n_i), n_i)$ (and equivalently, $\partial WCET(prev_{Use}(n_i), n_i)$).*

Proof. With regard to the proof of Proposition 1, this proof is straightforward, except that the replication process in backward movement is applied when n_i reaches a *Join* node and a backward movement is stopped when n_i reaches a *Branch* node (due to the control dependencies reflected in *MPDG*). □

According to Proposition 1 and Proposition 2, there might be two hotspots of type $\text{Hotspot}_{\text{Tail}}$ around a *Use* node n_i, so we have to decide whether move n_i backward or forward. More details on such a decision will be given in Section IV.

B. $Hotspot_{Sleep}$

Energy hotspot of type $\text{Hotspot}_{\text{Sleep}}$ is defined based on the time distance between two subsequent $Release(r)$ and $Acquire(r)$ statements in P. The time and energy overheads of these statements imply that putting r in the *Sleep* state is not always energy-efficient. More details are given below.

Suppose two subsequent pairs of $Acquire(r)$-$Release(r)$. The first $Release(r)$ and the second $Acquire(r)$ in the sequence consume $E_{Rel} + E_{Acq}$ amount of energy, which can be more than the additional energy that r consumes if it remains in the *Active* state by ignoring the releasing and re-acquiring of the module; where in case of such an ignorance, the two pairs of $Acquire(r)$-$Release(r)$ are converted to one pair.

This type of hotspots happens when the computational code running on *MCU* between the first $Release(r)$ and its subsequent $Acquire(r)$ is shorter than a threshold called *Lower Bound on Sleep Time (LBST)*, which is a lower bound on the execution time of the computational code to put r in the *Sleep* state. *LBST* is determined such that the energy of staying in the *Active* state for that amount of time be equal to the total energy of calling $Release(r)$, staying in the *Sleep* state for *LBST*, and calling $Acquire(r)$. Based on the parameters defined in Section II and simple algebraic manipulations, we find:

$$LBST = \frac{E_{Rel} + E_{Acq}}{Pow_A - Pow_S} \tag{5}$$

Proposition 3. *Suppose $\partial MCFG(A,B)$ as a piece of P, where A is a set of Rel nodes and B is a set of Acq nodes such that $prev_{Rel}(n_k)$ for every $n_k \in B$ is included in A and $next_{Acq}(n_k)$ for every $n_k \in A$ is included in B. Then, we have a $Hotspot_{Sleep}$ between A and B if $\partial WCET(A,B) \leq LBST$. The piece of program gets more energy-efficient if we remove Rel nodes belonging to A and Acq nodes belonging to B.*

Proof. Based on the definition of *LBST*, the energy consumption of r when it remains in the *Active* state for the longest path in $\partial MCFG(A,B)$ (by removing the nodes belonging to A and B from the path) is lower than when it is put in the *Sleep* state therein. Therefore, it is certain that the *Rel-Acq* pairs in all other paths (with lower execution times than $\partial WCET(A,B)$) cannot be affordable. □

C. $Hotspot_{Active}$

Energy hotspot of type $\text{Hotspot}_{\text{Active}}$ can be defined: 1) Between two subsequent $Acquire(r)$ and $Use(r,data)$ statements, and 2) between two subsequent $Use(r,data)$ and $Release(r)$ statements, based on their time distances in program P.

In the former case, if $Acquire(r)$ is called early (with a time distance greater than 0 to the subsequent $Use(r,data)$), r remains idle in the *Active* state, wasting some energy before the time that P reaches to $Use(r,data)$; this phenomena is called *Suboptimal Resource Binding* [12]. If we manipulate P by moving $Acquire(r)$ forward just beside the $Use(r,data)$ statement, then the energy hotspot is completely eliminated. In the latter case, if $Release(r)$ is called too late (with a time distance greater than 0 after the previous $Use(r,data)$), a similar situation takes place (r wastes some energy before the time that P reaches to $Release(r)$), which can be addressed by moving $Release(r)$ backward just after the $Use(r,data)$ statement.

Proposition 4. *Consider $\partial MCFG(n_i, next_{Use}(n_i))$ as a piece of P, where n_i is an Acq node. Then, we have a hotspot of type $Hotspot_{Active}$ if $\partial BCET(n_i, next_{Use}(n_i)) > 0$. The piece of program gets more energy-efficient if we move n_i forward towards minimizing $\partial BCET(n_i, next_{Use}(n_i))$ (and equivalently, $\partial WCET(n_i, next_{Use}(n_i))$).*

Proof. If $\partial BCET(n_i, next_{Use}(n_i)) > 0$, then r certainly spends some idle time in the *Active* state for all the paths in $\partial MCFG(n_i, next_{Use}(n_i))$. By moving n_i forward, it is placed closer to the elements of $next_{Use}(n_i)$, and thus, that piece of program, for all the paths, spends lower time in the *Active* state than before. If n_i reaches a *Branch* node during the forwarding, then it can be replicated and put at the start of all the paths after the *Branch* node, which can be treated as separate *Acq* nodes in the traverse of *MCFG*. Despite the control dependency constraints in *MPDG*, if n_i reaches a *Join* node during the forwarding and all other paths have an immediate *Acq* node before the *Join* node, n_i and those *Acq* nodes are merged as a single *Acq* node n_i, which is put after the *Join* node. Therefore, compared to $\partial MCFG(n_i, next_{Use}(n_i))$ before the movement, $\partial BCET(n_i, next_{Use}(n_i))$ is minimized, and thus, the piece of program gets more energy-efficient. □

Proposition 5. *Consider $\partial MCFG(prev_{Use}(n_i), n_i)$ as a piece of P, where n_i is a Rel node. Then, we have a hotspot of type $Hotspot_{Active}$ if $\partial BCET(prev_{Use}(n_i), n_i) > 0$. The piece of program gets more energy-efficient if we move n_i backward towards minimizing $\partial BCET(prev_{Use}(n_i), n_i)$ (and equivalently, minimizing $\partial WCET(prev_{Use}(n_i), n_i)$).*

Proof. Regarding the proof of Proposition 4, this proof is straightforward, except that the replication in backward movement is done when n_i reaches a *Join* node and the merging, if feasible, is performed when n_i reaches a *Branch* node. □

IV. Energy Hotspot Detection and Elimination

Suppose we have the *MCFG* and annotated *MPDG* of program P, along with the energy and time specifications of module (resource) r. Then, we need some static program analysis method which regulates the searches for hotspot detection and the acts for their elimination in a systematic traverse of *MCFG* with regard to the hotspot type. The result will then be an energy-improved program P', consistent to the dependencies and temporal restrictions reflected in *MPDG*.

A. Properties, Definitions and Rules

In this subsection, we provide several properties, definitions, and rules, necessary for proposing our energy hotspot detection and elimination algorithms in the next subsection.

Property 1. Neither an *Acq* node, nor a *Rel* node has a dependency with the *Comp* nodes around it; thus, either of them can be freely moved backward or forward between the *Comp* nodes. Eliminating a hotspot of type $\text{Hotspot}_{\text{Active}}$, which is done through moving an *Acq* node forward or a *Rel* node backward, has no impact on timing behavior of the *Use* nodes as well as that of P (i.e. C^- and C^+).

Property 2. Eliminating a $\text{Hotspot}_{\text{Sleep}}$, which is done through removing a *Rel-Acq* pair, shortens the paths involved with them, affecting c_i^- and c_i^+ of all *Use* nodes n_i in the paths, and possibly C^- and C^+.

Property 3. Eliminating a hotspot of type $\text{Hotspot}_{\text{Tail}}$ of a *Use* node n_i, which is done through moving the node n_i backward or forward, only affects c_i^- and c_i^+ of the node itself.

To check whether or not some manipulations on program P towards hotspot elimination violate the timing constraints of a *Use* node n_i, we need the following definitions.

Definition 6. Two *slacks* for each *Use* node n_i are defined regarding its time constraints d_i^- and d_i^+: $Slack_i^- = c_i^- - d_i^-$ and $Slack_i^+ = d_i^+ - c_i^+$.

Definition 7. Two *dependency slacks* for each *Use* node n_i are defined regarding its dependencies in *MPDG*: $DSlack_i^- = \partial BCET(A, n_i)$ and $DSlack_i^+ = \partial BCET(n_i, B)$, where $A = \{n_j : (n_j, n_i) \in E_{MPDG}\}$ and $B = \{n_j : (n_i, n_j) \in E_{MPDG}\}$.

The *Use* node n_i can be moved forward by at most $\tau_i^F = min(Slack_i^+, DSlack_i^+)$ and it can be moved backward by at

most $\tau_i^B = min(Slack_i^-, DSlack_i^-)$, to preserve the temporal correctness of P according to Section II-C.

To be able to evaluate the energy impact of transformation of P with regard to tail energy, we define two gain functions.

Definition 8. Suppose a *Use* node n_i is moved forward by $\tau \leq \tau_i^F$. A gain function $G_F(n_i, \tau)$, which is based on the tail energy consumed by n_i with respect to $next_{Use}(n_i)$, is as:

$$\begin{aligned} G_F(n_i,\tau) = & E_{Tail}(\partial WCET(n_i, next_{Use}(n_i))) \\ & - E_{Tail}(\partial WCET(n_i, next_{Use}(n_i)) - \tau) \\ & + E_{Tail}(\partial BCET(prev_{Use}(n_i), n_i)) \\ & - E_{Tail}(\partial BCET(prev_{Use}(n_i), n_i) + \tau) \end{aligned} \tag{6}$$

The first two terms in the right hand side of (6) calculate the minimum reduction in the tail energy wastage between n_i and $next_{Use}(n_i)$, and the last two terms calculate the maximum increase in the tail energy wastage between $prev_{Use}(n_i)$ and n_i by the forward movement. If $G_F(n_i, \tau) > 0$, the movement certainly (i.e. in all paths of $\partial MCFG(prev_{Use}(n_i), next_{Use}(n_i))$) causes reduction in the tail energy.

Definition 9. Suppose a *Use* node n_i is moved backward by $\tau \leq \tau_i^B$. A gain function $G_B(n_i, \tau)$, which is based on the tail energy consumed by n_i with respect to $prev_{Use}(n_i)$, is as:

$$\begin{aligned} G_B(n_i,\tau) = & E_{Tail}(\partial WCET(prev_{Use}(n_i), n_i)) \\ & - E_{Tail}(\partial WCET(prev_{Use}(n_i), n_i) - \tau) \\ & + E_{Tail}(\partial BCET(n_i, next_{Use}(n_i))) \\ & - E_{Tail}(\partial BCET(n_i, next_{Use}(n_i)) + \tau) \end{aligned} \tag{7}$$

The justification can be inspired by that of Definition 8.

In the remainder of this subsection, we provide a few rules to be used in the energy hotspot detection and elimination algorithms of the next subsection. The following two rules relate to $\text{Hotspot}_{\text{Tail}}$ with regard to node n_i:

Rule 1. If $G_F(n_i, \tau_i^F) > G_B(n_i, \tau_i^B)$, we can safely eliminate the hotspot via moving n_i forward by τ_i^F with no timing violation. The following updates are then needed:

$$\begin{aligned} c_i^- += \tau_i^F; c_i^+ += \tau_i^F; \\ Slack_i^- += \tau_i^F; Slack_i^+ -= \tau_i^F; \\ DSlack_i^- += \tau_i^F; DSlack_i^+ -= \tau_i^F. \end{aligned} \tag{8}$$

Rule 2. If $G_B(n_i, \tau_i^B) > G_F(n_i, \tau_i^F)$, we can safely eliminate the hotspot via moving n_i backward by τ_i^B with no timing violation. The following updates are then needed:

$$\begin{aligned} c_i^- -= \tau_i^B; c_i^+ -= \tau_i^B; \\ Slack_i^- -= \tau_i^B; Slack_i^+ += \tau_i^B; \\ DSlack_i^- -= \tau_i^B; DSlack_i^+ += \tau_i^B. \end{aligned} \tag{9}$$

Suppose *Rel* node n_i and *Acq* node n_j make a $\text{Hotspot}_{\text{Sleep}}$. Rules 3 to 5 relate to its elimination (see Proposition 3):

Rule 3. If for all *Use* nodes n_k in $\partial MCFG(n_j, End)$, $Slack_k^- \geq D_{Acq} + D_{Rel}$, then we can safely eliminate the hotspot with no temporal violation. The following updates are then needed for each node n_k mentioned above:

$$\begin{aligned} c_k^- -= (D_{Acq} + D_{Rel}); c_k^+ -= (D_{Acq} + D_{Rel}); \\ Slack_k^- -= (D_{Acq} + D_{Rel}); Slack_k^+ += D_{Acq} + D_{Rel}. \end{aligned} \tag{10}$$

Rule 4. Suppose that the condition of Rule 3 is not satisfied for a subset of *Use* nodes n_k in $\partial MCFG(n_j, End)$, denoted by χ. Then, we may be able to move forward each node $n_k \in \chi$ by $\tau_k = D_{Acq} + D_{Rel} - Slack_k^-$, to be able to eliminate the hotspot with no temporal violation. Therefore, the following conditions are checked for every *Use* node $n_k \in \chi$: 1) Whether $\tau_k \leq \tau_k^F$ or not, 2) whether $G_F(n_k, \tau_k) \geq 0$ or not; if the conditions are satisfied for all nodes of χ, we move forward every node $n_k \in \chi$ by τ_k, remove the hotspot of type $\text{Hotspot}_{\text{Sleep}}$, perform the updates of (10) for the nodes satisfying condition of Rule 3, and perform the following updates for the remaining nodes which all satisfy the conditions of this rule:

$$\begin{aligned} c_k^- -= Slack_k^-; c_k^+ -= Slack_k^-; \\ Slack_k^+ -= (D_{Acq} + D_{Rel} - Slack_k^-); \\ DSlack_k^+ -= (D_{Acq} + D_{Rel} - Slack_k^-); \\ DSlack_k^- += (D_{Acq} + D_{Rel} - Slack_k^-); \\ Slack_k^- = 0. \end{aligned} \tag{11}$$

Rule 5. We need to investigate the impact of eliminating the hotspot on C^-. Suppose $\Omega = \partial MCFG(n_j, End) \cap prev_{Use}(prev_{Rel}(End))$ as the set of last *Use* nodes of all the paths involved with the hotspot of type $\text{Hotspot}_{\text{Sleep}}$. For all *Use* nodes n_k in Ω, $\omega^- = min_k(c_k^- + \partial BCET(n_k, End))$ and $\omega^+ = max(c_k^+ + \partial WCET(n_k, End))$ are, respectively, the BCET and WCET among all the paths that involved with n_i and n_j. If $\omega^- - (D_{Acq} + D_{Rel}) \geq C^-$, then the hotspot can safely be eliminated; however, if $\omega^- - (D_{Acq} + D_{Rel}) < C^-$, the elimination can only be done if $\omega^- - (D_{Acq} + D_{Rel}) \geq D^-$. If done, the following updates are needed:

$$\begin{aligned} C^- = min(C^-, \omega^- - (D_{Acq} + D_{Rel})); \\ C^+ = \omega^+ - (D_{Acq} + D_{Rel}), \text{ if } C^+ == \omega^+. \end{aligned} \tag{12}$$

B. Detection and Elimination Algorithms

Our proposed algorithm, called Energy Hotspot Detection and Elimination (EHDE), is presented in Algorithm 1. Given a program P and the module specification *spec*, the algorithm returns the energy-improved program P' via removing the hotspots of type $\text{Hotspot}_{\text{Active}}$, $\text{Hotspot}_{\text{Tail}}$, and $\text{Hotspot}_{\text{Sleep}}$. It should be noted that our approach makes a modification in the program only if the energy consumption of all program paths get better than (or remain the same as) before. In fact, we do not suppose any information on the program runtime behavior with respect to execution of the program paths, namely we consider no knowledge that a path is executed rarely or frequently.

Algorithm 1 starts by constructing *MCFG* and *MPDG* of P (Line 1). It calculates the temporal characteristics (i.e. *slacks* and *dependency slacks*) of all *Use* nodes regarding restrictions reflected in *MPDG* (Line 2), and starts to traverse *MCFG* by

a Breadth-First Search approach; it continues to do so while there is an un-visited module-related node (Lines 3 and 4).

The algorithm detects and eliminates the corresponding hotspots depending on the type of visited node n_i (Lines 5 to 13). For an *Acq* node n_i, it calls Algorithm 2. For a *Use* node, it calls Algorithm 3. For a *Rel* node n_i, the algorithm calls Algorithm 2. For a $\text{Hotspot}_{\text{Sleep}}$ related to the *Rel* node n_i, however, we define A and B according to Proposition 3, and the algorithm analyzes the partial *MCFG* between A and B, and detects and removes a possible $\text{Hotspot}_{\text{Sleep}}$ through calling Algorithm 4. The process of removing the hotspots is performed until all the module-related nodes be visited in *MCFG*. Finally, the resulted *MCFG* is converted to the energy-improved program P' (Line 14).

Algorithm 1: EHDE Algorithm

Input : Program P, Module Specification *spec*
Output : Energy-Improved Program P'
1: Construct *MCFG* and annotated *MPDG* of P;
2: Calculate $Slack^-, Slack^+, DSlack^-, DSlack^+$ of *MPDG Use* nodes
3: **while** *MCFG* has an un-visited module-related node **do**
4: Let n_i be the next visited module-related node in *MCFG*;
5: **if** $type_i = Acq$ **then**
6: $\text{Hotspot}_{\text{Active}}\text{DE}(n_i)$;
7: **else if** $type_i = Use$ **then**
8: $\text{Hotspot}_{\text{Tail}}\text{DE}(n_i)$;
9: **else if** $type_i = Rel$ **then**
10: $\text{Hotspot}_{\text{Active}}\text{DE}(n_i)$;
11: $B = next_{Acq}(n_i)$;
12: $A = \{prev_{Rel}(n_k) | n_k \in B\}$;
13: $\text{Hotspot}_{\text{Sleep}}\text{DE}(A,B)$;
14: Convert *MCFG* to the energy-improved program P';
15: **return** P';

Algorithm 2 shows the procedure of detection and elimination of $\text{Hotspot}_{\text{Active}}$ from *MCFG*. It gets n_i, and removes a $\text{Hotspot}_{\text{Active}}$ related to n_i if exists. If n_i is an *Acq* node, the algorithm checks the partial BCET between n_i and $next_{Use}(n_i)$, and calls ForwardAcq(n_i) if there is some distance to the *Use* node (see Proposition 4), which moves n_i forward in *MCFG*, and if n_i reaches a *Branch* node in the meanwhile, replicates it and puts the replicas at the start of all paths after the *Branch*; further, the algorithm adds all the replicas to the set of un-visited nodes, to be considered in further steps of Algorithm 1. If n_i reaches a *Join* node n_j in the forward movement, then it checks whether all other paths end to an immediate *Acq* node n_k just before n_j; if so, n_i and all the nodes n_k are merged in a single *Acq* node put just after n_j, which is added to the set of un-visited nodes, to be considered further by Algorithm 1 (Lines 1 to 6).

Otherwise, if n_i is a *Rel* node, the algorithm checks the partial BCET between $prev_{Use}(n_i)$ and n_i, and calls BackwardRel(n_i) if there is some distance between $prev_{Use}(n_i)$ and n_i (see Proposition 5), which moves n_i backward in *MCFG*, and if n_i reaches a *Join* node in the meanwhile, replicates it and puts the replicas at the end of all paths before the *Join* node; further, the algorithm adds the replicas to the set of un-visited nodes, to be considered further by Algorithm 1. If n_i reaches a *Branch* node n_j in the backward movement, then the algorithm checks whether all other paths begin with an immediate *Rel* node n_k just after n_j; if so, n_i and all the nodes n_k are merged in a single *Rel* node put just before n_j, which is added to the set of un-visited nodes (Lines 7 to 12).

Algorithm 2: $\text{Hotspot}_{\text{Active}}\text{DE}$

Input : Node n_i
1: **if** $type_i$ is *Acq* **then**
2: **if** $\partial BCET(n_i, next_{Use}(n_i)) > 0$ **then**
3: ForwardAcq(n_i);
4: **if** All branches before *Join* end to *Acq* **then**
5: Merge them into an *Acq* and move it after *Join*;
6: Add the *Acq* node to the set of un-visited nodes;
7: **else if** $type_i$ is *Rel* **then**
8: **if** $\partial BCET(prev_{Use}(n_i), n_i) > 0$ **then**
9: BackwardRel(n_i);
10: **if** All branches after *Branch* start with *Rel* **then**
11: Merge them into a *Rel* and move it before *Branch*;
12: Add the *Rel* node to the set of un-visited nodes;

Algorithm 3 gives the procedure of detection and elimination of $\text{Hotspot}_{\text{Tail}}$ from *MCFG*. It gets n_i, and checks whether there exists a $\text{Hotspot}_{\text{Tail}}$ related to n_i; if so, it removes the hotspot from *MCFG*. The algorithm starts by checking the partial BCET between n_i and $next_{Use}(n_i)$, and between $prev_{Use}(n_i)$ and n_i to detect potentials related to $\text{Hotspot}_{\text{Tail}}$ (see Propositions 1 and 2). Then, it calculates τ_i^F and τ_i^B as the possible time distances for forward and backward movements respectively (Lines 2 and 3). According to Rules 1 and 2, the algorithm decides about the direction of movement, and removes the corresponding hotspot (Lines 4 to 10). It should be noted that when $G_F(n_i, \tau_i^F) = G_B(n_i, \tau_i^B)$, the algorithm chooses to forward n_i, bringing it closer to the next *Use* node n_{i+1}, which has not been visited yet, leaving some chance for further tail energy removal. The temporal characteristics of n_i after the hotspot removal are updated according to (8) and (9) for forward and backward movements, respectively.

ForwardUse(n_i, τ_i^F) moves n_i forward in *MCFG* if there is some distance to $next_{Use}(n_i)$, considering the dependency and timing constraints. Meanwhile, if n_i reaches a *Branch* node, the *Use* node is replicated and put at the start of all paths after *Branch*, and all the replicas are added to the set of un-visited nodes, to be considered further by Algorithm 1. We stop moving a *Use* node forward if it reaches a *Join* node, due to the dependency constraints of *MPDG* (see Section II-B). Similar discussions can be done on BackwardUse(n_i, τ_i^B) except that we replicate the *Use* node n_i if it reaches a *Join* node, and we stop moving it backward if it reaches a *Branch* node.

Algorithm 4 provides the procedure of detection and elimination of $\text{Hotspot}_{\text{Sleep}}$ from *MCFG*. It gets the set of *Rel* nodes A and the set of *Acq* nodes B, and starts by calculating *LBST* based on (5) (Line 1). Then, it checks the partial WCET between A and B for a potential of energy hotspot (see Proposition 3). In the next step, the algorithm builds Ω, calculates ω^-, and checks the condition of Rule 5

Algorithm 3: Hotspot$_{\text{Tail}}$DE

Input : *Use* node n_i

1: **if** $\partial BCET(n_i, next_{Use}(n_i)) > 0 \wedge \partial BCET(prev_{Use}(n_i), n_i) > 0$ **then**
2: $\tau_i^F = min(Slack_i^+, DSlack_i^+)$;
3: $\tau_i^B = min(Slack_i^-, DSlack_i^-)$;
4: Calculate $G_F(n_i, \tau_i^F)$ and $G_B(n_i, \tau_i^B)$ of *spec*;
5: **if** $G_F(n_i, \tau_i^F) \geq G_B(n_i, \tau_i^B)$ **then**
6: ForwardUse(n_i, τ_i^F);
7: Update temporal characteristics of n_i acc. Rule 1;
8: **else if** $G_B(n_i, \tau_i^B) > G_F(n_i, \tau_i^F)$ **then**
9: BackwardUse(n_i, τ_i^B);
10: Update temporal characteristics of n_i acc. Rule 2;

to investigate whether eliminating the hotspot respects D^- or not (Lines 3 to 5). According to Rules 3 and 4, the algorithm checks whether all *Use* nodes in $\partial MCFG(B, End)$ have sufficient slack (i.e. $Slack_k^-$ for a *Use* node n_k) for the hotspot removal or not; if so, it safely removes A and B from *MCFG*, updates *MPDG* with regard to the removed module-related nodes, and updates the temporal characteristics according to (10) and (12) (Lines 6 to 8). Otherwise, the algorithm tries to move forward some *Use* nodes which do not satisfy the previous condition, denoted by a subset χ, by $\tau_k = D_{Acq} + D_{Rel} - Slack_k^-$, to possibly use the slacks after the *Use* nodes (namely $\tau_k^F = min(Slack_k^+, DSlack_k^+)$) and make the hotspot removal feasible, however after making sure that the forward movement is energy-efficient (via checking $G_F(n_k, \tau_k)$). If feasible, the algorithm safely removes A and B from *MCFG*, updates *MPDG* with regard to the removed module-related nodes, and updates the temporal characteristics of the *Use* nodes according to (10), (11) and (12) (Lines 10 to 15).

Algorithm 4: Hotspot$_{\text{Sleep}}$DE

Input : Set of *Rel* nodes A, Set of *Acq* nodes B

1: Calculate *LBST* of *spec*;
2: **if** $\partial WCET(A, B) \leq LBST$ **then**
3: $\Omega = \partial MCFG(B, End) \cap prev_{Use}(prev_{Rel}(End))$;
4: $\omega^- = min_k(c_k^- + \partial BCET(n_k, End))$ for all $n_k \in \Omega$;
5: **if** $\omega^- - (D_{Acq} + D_{Rel}) \geq D^-$ **then**
6: **if** all $n_k \in \partial MCFG(B, End)$ satisfy $Slack_k^- \geq D_{Acq} + D_{Rel}$ **then**
7: Remove A and B from *MCFG*, and update *MPDG* accordingly;
8: Update temporal characteristics acc. Rules 3 and 5;
9: **else**
10: $\chi = \{n_k | type_k = Use, Slack_k^- < D_{Acq} + D_{Rel}\}$;
11: $\tau_k = D_{Acq} + D_{Rel} - Slack_k^-$;
12: **if** all $n_k \in \chi$ satisfy $(\tau_k \leq \tau_k^F \wedge G_F(n_k, \tau_k) \geq 0)$ **then**
13: ForwardUse(n_k, τ_k) for all $n_k \in \chi$;
14: Remove A and B from *MCFG*, and update *MPDG* accordingly;
15: Update temporal characteristics acc. Rules 4 and 5;

Notice that, without loss of generality, the timing analysis for hotspot elimination is based on a simple architectural model of typical microcontrollers used in some popular timing analyzers (including the tool we employed in Section V) which ignore some profound effects like cache in their analysis. Extensions to more complex architectures that are subject to timing anomalies are possible, but need either more elaborate timing analysis methods as used for the cache, or repeated timing analysis after each hotspot elimination.

V. Evaluation

This section describes the simulation setup, simulation results and comparisons to other methods, and some discussions.

A. Simulation Setup

Hardware Setup and Simulation Tools: The embedded hardware considered for the evaluations is an AT91SAM7X256 MCU with an ARM7TDMI processor, 64 KB of SRAM, and 256 KB of Flash. It is equipped with a LTE BG96 module, which is widely used for wireless communication in embedded systems. We used MEET [23] (based on SimpleScalar [24]) to simulate the execution time and energy consumption of computational instructions. The energy, time, and power specifications of the module with regard to its states and the state transitions have been extracted from the datasheet [17] (see Table I). For the tail energy, thus, we suppose one state with an average power of Pow_{Tail} and an average delay of D_{Tail}. The time and energy of a given program obtained using MEET and the specifications of BG96 with regard to the driver calls are combined together to simulate the time and energy behaviors of the deeply embedded system.

Time Analyzer Tool: To extract the program CFG and PDG, identify its execution paths, and calculate the partial WCETs/BCETs, we used SWEET [25] program flow analysis tool. Further, for calculating the partial WCETs/BCETs, the partial graphs are converted to distinct functions.

Test Programs and Program Configurations: Receiving data, processing them, and sending the processed data are the well-known operations in most deeply embedded devices in the context of IoT [26]. Thus, the test programs should comprise both computational and module-related statements. We use computational parts from MiBench [16] benchmark suite and augment them with module-related statements, under different configurations. We used five application benchmarks of *rijndael* (encode and decode), *basicmath*, *patricia*, *jpeg* (encode and decode), and *susan* (corners, edges and smoothing), as the computational parts of our test programs, shown respectively as Programs 1, 2, 3, 4, and 5.

We consider four different configurations of CONF 1, CONF 2, CONF 3, and CONF 4, as in Table II, for augmenting the module-related statements (nodes). The configurations cover a broad range of sparse to dense module-related statements through randomly (according to the uniform distribution) injecting the statements in the program to create variety of energy hotspots contributing to the total program energy consumption. The parameters of a configuration are the minimum and maximum numbers of *Acq-Rel* pairs and the *Use* nodes between such a pair. We performed flow analysis using SWEET to find different program execution paths $p_k \in P$, to

TABLE II: Different configurations for augmenting computational parts with module-related statements (nodes).

Parameters	Configurations			
	CONF 1	CONF 2	CONF 3	CONF 4
Min / Max $Acq-Rel$ pairs	1 / 3	4 / 6	7 / 9	10 / 12
Min / Max *Use* between every $Acq-Rel$	1 / 5	6 / 10	11 / 15	16 / 20

TABLE III: The offline static analysis time for EHDE.

Item	Program 1 *rijndael*	Program 2 *basicmath*	Program 3 *patricia*	Program 4 *jpeg*	Program 5 *susan*
Code size (instruction count)	1.65×10^8	10^9	10^9	1.53×10^9	2.31×10^9
Static analysis time	<10 min	<20 min	<20 min	<40 min	< 1 hour

randomly inject the module-related nodes of the configurations among the *Comp* nodes. When a program execution path is augmented, the remaining paths are treated to preserve the program logical correctness (e.g. to have no resource leak).

Time Restrictions: D^- and D^+ are generated randomly in the range of $[\frac{1}{2}C^-, \frac{9}{10}C^-]$ and $[\frac{11}{10}C^+, \frac{3}{2}C^+]$, respectively, i.e. the slacks are at least 10% and at most 50% of the corresponding execution times of C^- and C^+. Also, for time constraint assignment to the *Use* nodes, we performed as $d_i^- = c_i^- \times \frac{D^-}{C^-}$ and $d_i^+ = c_i^+ \times \frac{D^+}{C^+}$, $\forall n_i \in N, type_i = Use$ (inspired from [27]), although any valid time restrictions can be assigned.

Compared Methods: We compare the results of three methods: 1) EHDE, which is our proposed method concerning the hotspots of types $\text{Hotspot}_{\text{Active}}$ and $\text{Hotspot}_{\text{Sleep}}$ and $\text{Hotspot}_{\text{Tail}}$ for real-time deeply embedded systems; 2) HR [19], which is a static analysis method only concerning the hotspots of type $\text{Hotspot}_{\text{Sleep}}$ for non real-time embedded systems; 3) T-DPM [28], which is a timeout-based dynamic power management (DPM) technique (a technique with no program modification), which switches the peripheral module to a lower power state based on a timeout policy, and returns it to a higher power state whenever its functionality is needed, while the switching does not result in violating $C^+ \leq D^+$.

Finally, Table III shows the program sizes and the average static analysis times of EHDE, where the analyses and simulations were run on a system with an Intel CoreTM i7-4720HQ processor.

B. Simulation Results

Suppose Programs 1 to 5 under configurations CONF 1 to CONF 4, constituting 20 test programs. Due to huge number of program paths, and to estimate the performance improvement of the methods, we randomly selected some paths according to the provided input data set for every test program, augmented with the paths corresponding to C^- and C^+. Then, we averaged the results over the paths of each test program. Note that our proposed method EHDE never makes a transformation if the energy consumption of a program path may get worse or a time constraint may be violated. Fig. 3 illustrates the total energy improvement resulted from applying EHDE with respect to the original test program as well as the contribution of the hotspot types in the reduced energy. As it can be seen, the best-case improvement, with 20% energy reduction, relates to Program 5 in CONF 4, and the worst-case improvement, with 4.7% energy reduction, relates to Program 1 in CONF 1.

On average, eliminating the hotspots of types $\text{Hotspot}_{\text{Tail}}$, $\text{Hotspot}_{\text{Sleep}}$ and $\text{Hotspot}_{\text{Active}}$ have had 43%, 34% and 23% contributions on the energy reductions, respectively. As it can be seen, by increasing the number of module-related nodes from CONF 1 to CONF 4, the contribution of $\text{Hotspot}_{\text{Sleep}}$ shows a decreasing trend, while that of $\text{Hotspot}_{\text{Tail}}$ exhibits an inverse trend. Regarding Table II, the number of *Use* nodes between each *Acq-Rel* pair increases by an average of 5 nodes per configuration, which directly increases the chance of eliminating more hotspots of type $\text{Hotspot}_{\text{Tail}}$. However, although the number of *Rel-Acq* pairs, and thus the number of hotspots of type $\text{Hotspot}_{\text{Sleep}}$, increases from CONF 1 to CONF 4, and the reduced energy by eliminating a $\text{Hotspot}_{\text{Sleep}}$ is about 4 times that of a $\text{Hotspot}_{\text{Tail}}$, eliminating many hotspots of type $\text{Hotspot}_{\text{Sleep}}$ are not possible due to the time restrictions of the program, namely $D^- \leq C^-$ and $d_i^- \leq c_i^-$ for further *Use* nodes n_i. Regarding the contribution of $\text{Hotspot}_{\text{Active}}$ on the energy reduction, no time restrictions need to be considered and the elimination is straightforward; however, by increasing the number (and thus, density) of the module-related nodes, the energy waste of this type decreases on average; furthermore, BG96 is not too power-hungry in the *Active* state.

Fig. 4 gives more details on the results and compares EHDE with HR and T-DPM. It reports the energy improvements with respect to the original test programs, separated for the MCU and the module after applying the algorithms. It also reports the values of C^+, C^-, and average execution time (*avgET*) of the test programs after applying the methods, normalized with respect to C^+ of the corresponding original test program.

According to Fig. 4, as expected, most energy improvements of EHDE are from the module, while minor improvements are due to the MCU and the reduced *Idle* times of eliminating hotspots of type $\text{Hotspot}_{\text{Sleep}}$. Such improvements are higher for HR because it does not consider time restrictions, and thus, eliminates more hotspots of type $\text{Hotspot}_{\text{Sleep}}$. Moreover, T-DPM works inversely in some test programs by consuming extra MCU *Idle* energy, which is due to additional power

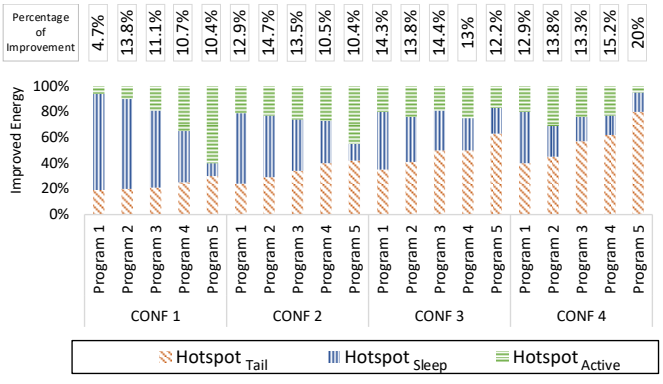

Fig. 3: The total energy improvements by EHDE and the contribution of different types of energy hotspots in them.

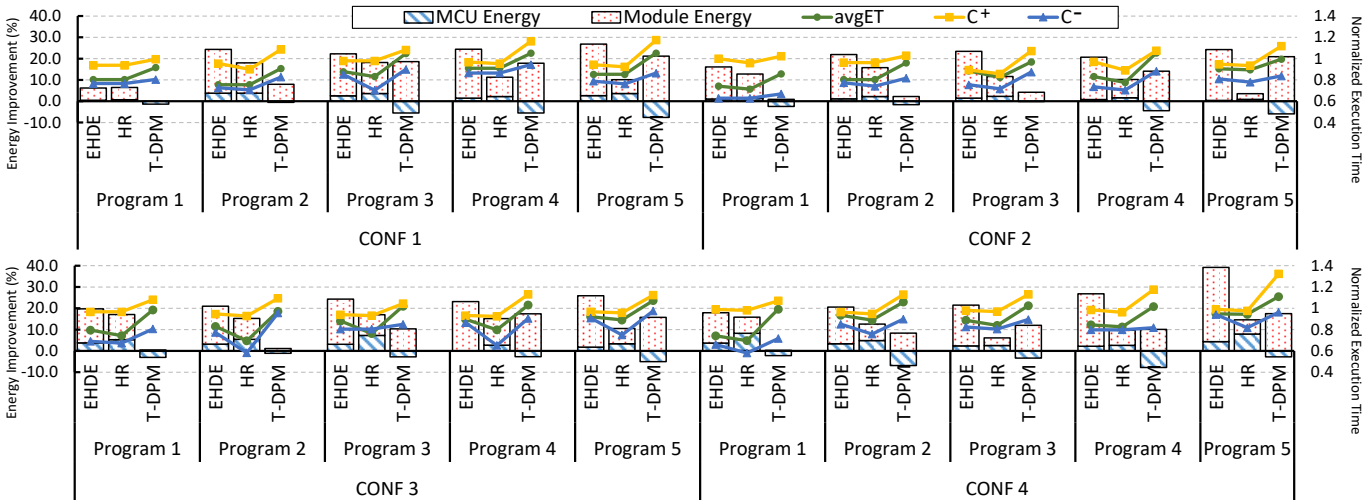

Fig. 4: The energy improvement of EHDE, HR, and T-DPM, separated for the module and MCU, with respect to the original test programs, and the values of C^+, C^- and $avgET$, normalized to C^+ of the corresponding original programs.

state switches at the system level, imposing extra waiting time overheads on the MCU. However, from the module's point of view, EHDE improves the energy consumption much better than HR or T-DPM because it detects and eliminates hotspots of $\text{Hotspot}_{\text{Tail}}$ and $\text{Hotspot}_{\text{Active}}$ types, and it does not waste the power of the intervals when T-DPM waits for timeout and avoids the increased switching of the module's power state.

When the module forces a power state change through T-DPM, C^-, C^+, and $avgET$ increase, which can be seen in Fig. 4; however, since we are discussing in the context of real-time systems, we have considered the time constraint of $C^+ \leq D^+$, so that we stop the timeout-based power switching when it could lead to a time violation. Inversely, C^-, C^+, and $avgET$ are only decreased in HR (because of $\text{Hotspot}_{\text{Sleep}}$), which may violate the time constraint of $D^- \leq C^-$.

Even though $C^+ \leq D^+$ has been respected when applying T-DPM, violation of the time restrictions of *Use* nodes is still possible (i.e. to have $c_i^+ > d_i^+$ for *Use* node n_i). In HR, however, in addition to the possibility of violating $C^- \geq D^-$, violation of the time restrictions of *Use* nodes is still possible (i.e. to have $c_i^- < d_i^-$ for node n_i). In contrast, EHDE respects all the time restrictions. We now scrutinize this via a measure of miss-ratio, defined as the worst-case fraction of *Use* nodes with time violations in the modified test program executions.

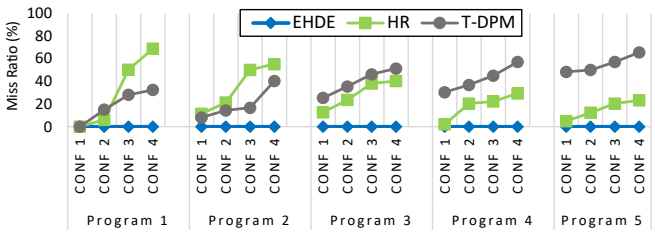

Fig. 5: The worst-case miss-ratios of $Use(r, data)$ statements.

Fig. 5 gives the miss-ratios for EHDE, HR, and T-DPM, where the former has a constant miss-ratio of 0 percent. The miss-ratio of the programs rise up when the number of module-related statements (nodes) are increased. In fact, further energy improvements by HR and T-DPM lead to more timing violations of the *Use* nodes.

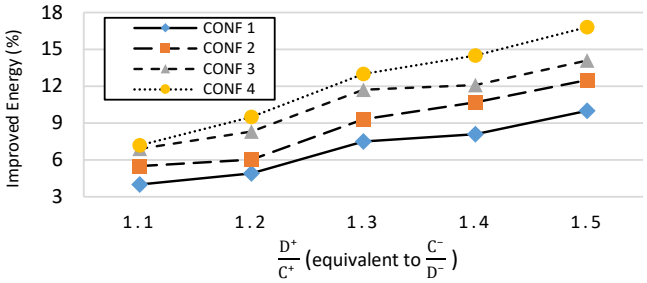

Fig. 6: The energy impact of slacks on EHDE.

By the increase of program size from Program 1 to Program 5, the time distances between the consecutive *Rel* and *Acq* nodes increase as well; therefore, elimination of hotspots of type $\text{Hotspot}_{\text{Sleep}}$ in HR becomes less possible. As it can be seen in Fig. 5, it causes the miss-ratio of HR to be reduced from Program 1 to Program 5. These further distances, however, cause miss-ratios of T-DPM to rise up, because the timeout-based power state changes are more likely to be done. Overall, we see the worst-case miss-ratio of about 68.8% for HR, and that of about 65.5% for T-DPM.

Finally, Fig. 6 investigates the effect of program slacks on the average energy improvements. Accordingly, more energy improvements are possible when the slacks are increased; i.e. the less time restriction the program has, the more energy hotspots can be eliminated by EHDE. Further, in the presence of more module-related statements, energy hotspots are more probable in the program, so that EHDE is more effective.

VI. Related Work

In this section, we review the literature of program energy consumption from different perspectives. Some of the studies have focused on CPU, and some of them have considered peripheral devices. From another perspective, some studies take care of the program code details, and some others take care of system-level issues.

Program energy defect detection: The program energy defect analysis methods can be categorized into two groups, none of which is mature yet: Testing [12], [29], [30] and static analysis [13], [14], [15], [31]. Some major issues of energy testing are the need to program execution and energy measurement to ascertain the existence of energy defects, whereas they are not inclusive. Static analysis methods do not have such issues. The literature of static analysis methods mostly focus on revealing the energy bugs, and energy hotspots have less been investigated. In [19], some sort of energy hotspot, i.e. $\text{Hotspot}_{\text{Sleep}}$, in the context of non real-time embedded systems has been studied. It also discusses the energy wastage in the CPU idle state when the driver calls of the peripheral module are non-blocking. The current work, however, discusses program energy hotspots of real-time embedded systems with a novel look to the relationship between WCET/BCET of the program pieces and the energy and time overheads of the module driver calls, and provides algorithms to automatically detect and eliminate such hotspots.

Program energy optimization: There are remarkable studies on program energy optimization and the corresponding tools, as surveyed in [8]. The authors of [32], [33] investigated the impact of techniques like algorithmic optimization, loop transformation, and function inlining on power consumption of different embedded platforms. Honig et al. [34] proposed a proactive energy-aware programming approach, called *PEEK*, to give the energy optimization hints directly to the developer during the software development process based on executing the program at different power states considering run-time requirements. Georgiou et al. [35] realized energy estimation and optimization at the compiler intermediate representation level for deeply embedded systems. However, these studies focus on the CPU energy consumption.

Program energy evaluation: Several studies have focused on modeling and evaluation of the peripheral module energy consumption at the program level. As an illustration, Cherifi et al. [36] proposed a method to generate an automata-based energy consumption model of the modules, based on run-time energy measurements. Berthou et al. [37] proposed a power model of low-power embedded systems equipped with peripheral modules, for using in a cycle-accurate simulation of the program execution. Authors of [38] and [5] developed two I/O energy profiling frameworks for embedded systems (*EPROM*) and mobile devices (*EPROF*), respectively. Although these studies have sketched a clear view of how the energy is consumed by the peripheral modules, they only help the developer to get a more accurate comprehension of them, and they have no focus on program level energy improvement.

Dynamic power management (DPM) techniques: DPM is a significant energy management technique to deal with the energy efficiency issue of hardware components, including CPU and peripheral modules [39]. It puts the component into different power states depending on the situation and the required functionality. DPM approaches for the peripheral modules [28], [40], [41] usually work on the basis of timeout or predictions to change the power states; as a result, the module consumes energy in vain until a certain threshold is passed, and sometimes the decisions might negatively impact the module or system energy consumption. In the current study, however, we perform at the program level, and guarantee that the modifications do not worsen the energy consumption. More precisely, we target some energy hotspots, and remove the root-cause of them in deeply embedded programs by some deterministic static analysis approaches.

VII. Conclusion

Real-time deeply embedded devices are designed to provide a wide variety of applications in the context of IoT. Thus, non-functional aspects, including energy consumption, are quite important considerations. This paper discusses this concern for specific types of energy hotspots through a static program analysis method, which its program transformations have no negative energy impact on any program path and respect the real-time constraints of the embedded software. The simulation results confirm the efficacy of the proposed method. Extending the work to various types of energy hotspots under more sophisticated theoretical aspects as well as implementing it as a comprehensive toolchain is of supposed future work.

References

[1] Umesh Balaji Kothandapani Ramesh, Severine Sentilles, and Ivica Crnkovic. Energy management in embedded systems: Towards a taxonomy. In *First International Workshop on Green and Sustainable Software (GREENS)*, pages 41–44. IEEE, 2012.

[2] Ali Abbasi, Jos Wetzels, Thorsten Holz, and Sandro Etalle. Challenges in designing exploit mitigations for deeply embedded systems. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 31–46. IEEE, 2019.

[3] AspenCore Global Media. Embedded markets study, integrating iot and advanced technology designs, application development and processing environments. *Electronic Engineering Times available at Embedded.com*, 2019.

[4] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering developers to estimate app energy consumption. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 317–328, 2012.

[5] Abhinav Pathak, Y Charlie Hu, and Ming Zhang. Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 29–42, 2012.

[6] Di Zhang, Yaoxue Zhang, Yuezhi Zhou, and Hao Liu. Leveraging the tail time for saving energy in cellular networks. *IEEE Transactions on Mobile Computing*, 13(7):1536–1549, 2013.

[7] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E Hassan. What do programmers know about software energy consumption? *IEEE Software*, 33(3):83–89, 2015.

[8] Cuijiao Fu, Depei Qian, Tianming Huang, and Zhongzhi Luair. A survey on energy consumption oriented program test and analysis technology and tools. In *Proceedings of the International Conference on Scientific Computing (CSC)*, pages 73–79, 2019.

[9] Kshirasagar Naik. *A survey of software based energy saving methodologies for handheld wireless communication devices*. Department of Electrical and Computer Engineering, University of Waterloo, 2010.

[10] Jan Jelschen, Marion Gottschalk, Mirco Josefiok, Cosmin Pitu, and Andreas Winter. Towards applying reengineering services to energy-efficient applications. In *16th European Conference on Software Maintenance and Reengineering*, pages 353–358. IEEE, 2012.

[11] Kyriakos Georgiou, Samuel Xavier-de Souza, and Kerstin Eder. The iot energy challenge: A software perspective. *IEEE Embedded Systems Letters*, 10(3):53–56, 2017.

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

[13] Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury. Energypatch: Repairing resource leaks to improve energy-efficiency of android apps. *IEEE Transactions on Software Engineering*, (5):470–490, 2017.

[14] Haowei Wu, Shengqian Yang, and Atanas Rountev. Static detection of energy defect patterns in android applications. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 185–195, 2016.

[15] Hao Jiang, Hongli Yang, Shengchao Qin, Zhendong Su, Jian Zhang, and Jun Yan. Detecting energy bugs in android apps using static analysis. In *International Conference on Formal Engineering Methods*, pages 192–208. Springer, 2017.

[16] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization*, pages 3–14. IEEE, 2001.

[17] Quectel. BG96 LTE Module. URL: https://www.quectel.com/product/lpwa-bg96-cat-m1-nb1-egprs, last accessed on 05/26/2021.

[18] Atmel (Microchip Technology). AT91SAM7X256 Micro-contorller unit. URL: https://www.microchip.com/wwwproducts/en/AT91SAM7X256, last accessed on 05/26/2021.

[19] Mohsen Shekarisaz, Fatemeh Talebian, Marjan Jabariani, Farzad Mehri, Fathiyeh Faghih, and Mehdi Kargahi. Program energy-hotspot detection and removal: A static analysis approach. In *CSI/CPSSI International Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*, pages 1–8. IEEE, 2020.

[20] Rabab Bouziane, Erven Rohou, and Abdoulaye Gamatié. Energy-efficient memory mappings based on partial wcet analysis and multi-retention time stt-ram. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, pages 148–158, 2018.

[21] Yusuke Sabi, Yoshiki Higo, and Shinji Kusumoto. Rearranging the order of program statements for code clone detection. In *2017 IEEE 11th International Workshop on Software Clones (IWSC)*, pages 1–7. IEEE, 2017.

[22] Haisang Wu, Binoy Ravindran, E Douglas Jensen, and Peng Li. Time/utility function decomposition techniques for utility accrual scheduling algorithms in real-time distributed systems. *IEEE Transactions on Computers*, 54(9):1138–1153, 2005.

[23] Mostafa Bazzaz, Mohammad Salehi, and Alireza Ejlali. An accurate instruction-level energy estimation model and tool for embedded systems. *IEEE Transactions on Instrumentation and Measurement*, 62(7):1927–1934, 2013.

[24] Doug Burger and Todd M Austin. The simplescalar tool set. *ACM SIGARCH computer architecture news*, 25(3):13–25, 1997.

[25] SWEET (SWEdish Execution Time tool). URL: http://www.mrtc.mdh.se/projects/wcet/sweet.html/, last accessed on 05/26/2021.

[26] Farzad Samie, Lars Bauer, and Jörg Henkel. Iot technologies for embedded computing: A survey. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 1–10. IEEE, 2016.

[27] Faeze Eshragh and Mehdi Kargahi. Analytical architecture-based performability evaluation of real-time software systems. *Journal of Systems and Software*, 86(1):233–246, 2013.

[28] Hakduran Koc and Pranitha P Madupu. Optimizing energy consumption in cyber physical systems using multiple operating modes. In *8th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 520–525. IEEE, 2018.

[29] Reyhaneh Jabbarvand, Jun-Wei Lin, and Sam Malek. Search-based energy testing of android. In *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1119–1130. IEEE, 2019.

[30] Farzaneh Azimian, Fathiyeh Faghih, Mehdi Kargahi, and SM Mahdi Mirdehghan. Energy metamorphic testing for android applications. In *IEEE 30th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC Workshops)*, pages 1–6. IEEE, 2019.

[31] Chang Hwan Peter Kim, Daniel Kroening, and Marta Kwiatkowska. Static program analysis for identifying energy bugs in graphics-intensive mobile apps. In *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 115–124, 2016.

[32] Alexander Chemeris, Dmitri Lazorenko, and Sergey Sushko. Influence of software optimization on energy consumption of embedded systems. In *Green IT Engineering: Components, Networks and Systems Implementation*, pages 111–133. Springer, 2017.

[33] David A Ortiz and Nayda G Santiago. Impact of source code optimizations on power consumption of embedded systems. In *6th International IEEE Northeast Workshop on Circuits and Systems and TAISA Conference*, pages 133–136. IEEE, 2008.

[34] Timo Hönig, Heiko Janker, Christopher Eibel, Oliver Mihelic, and Rüdiger Kapitza. Proactive energy-aware programming with PEEK. In *Conference on Timely Results in Operating Systems (TRIOS)*, 2014.

[35] Kyriakos Georgiou, Steve Kerrison, Zbigniew Chamski, and Kerstin Eder. Energy transparency for deeply embedded programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(1):1–26, 2017.

[36] Nadir Cherifi, Thomas Vantroys, Alexandre Boe, Colombe Herault, and Gilles Grimaud. Automatic inference of energy models for peripheral components in embedded systems. In *IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 120–127. IEEE, 2017.

[37] Gautier Berthou, Kevin Marquet, Tanguy Risset, and Guillaume Salagnac. Accurate power consumption evaluation for peripherals in ultra low-power embedded systems. In *2020 Global Internet of Things Summit (GIoTS)*, pages 1–6. IEEE, 2020.

[38] Shiao-Li Tsao, Cheng-Kun Yu, and Yi-Hsin Chang. Profiling energy consumption of i/o functions in embedded applications. In *International Conference on Architecture of Computing Systems*, pages 195–206. Springer, 2013.

[39] Bishop Brock and Karthick Rajamani. Dynamic power management for embedded systems [soc design]. In *IEEE International [Systems-on-Chip] SOC Conference, 2003. Proceedings.*, pages 416–419. IEEE, 2003.

[40] Taliver Heath, Eduardo Pinheiro, Jerry Hom, Ulrich Kremer, and Ricardo Bianchini. Code transformations for energy-efficient device management. *IEEE Transactions on Computers*, 53(8):974–987, 2004.

[41] Peng Rong and Massoud Pedram. Determining the optimal timeout values for a power-managed system based on the theory of markovian processes: offline and online algorithms. In *Proceedings of the Design Automation & Test in Europe Conference*, volume 1, pages 1–6. IEEE, 2006.