To appear in: Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS'15)
Lund, Sweden, 7–10 July 2015

Worst-Case Energy Consumption Analysis for Energy-Constrained Embedded Systems

Peter Wägemann, Tobias Distler, Timo Hönig, Heiko Janker, Rüdiger Kapitza[1], and Wolfgang Schröder-Preikschat
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) and [1]TU Braunschweig

***Abstract*—The fact that energy is a scarce resource in many embedded real-time systems creates the need for energy-aware task schedulers, which not only guarantee timing constraints but also consider energy consumption. Unfortunately, existing approaches to analyze the worst-case execution time (WCET) of a task usually cannot be directly applied to determine its worst-case energy consumption (WCEC) due to execution time and energy consumption not being closely correlated on many state-of-the-art processors. Instead, a WCEC analyzer must take into account the particular energy characteristics of a target platform.**

In this paper, we present *0g*, a comprehensive approach to WCEC analysis that combines different techniques to speed up the analysis and to improve results. If detailed knowledge about the energy costs of instructions on the target platform is available, our tool is able to compute upper bounds for the WCEC by statically analyzing the program code. Otherwise, a novel approach allows *0g* to determine the WCEC by measurement after having identified a set of suitable program inputs based on an auxiliary energy model, which specifies the energy consumption of instructions in relation to each other. Our experiments for three target platforms show that *0g* provides precise WCEC estimates.

I. Introduction

In many embedded real-time systems energy is a scarce resource: either because they completely lack mechanisms to recharge their batteries after having been deployed, as it is the case for energy-constrained systems (e.g., underwater sensor nodes), or because the amount of energy these systems harvest can only prolong their lifetime but cannot guarantee infinite operation (e.g., solar-powered sensor nodes). Several authors have argued that when energy plays such a key role, it should also become an important factor when it comes to making scheduling decisions [1], [2], [3]. That is, in addition to ensuring that a task meets its timing deadline, energy-aware task schedulers also consider whether or not there is enough energy available in the system for the task to complete execution. As a consequence, besides knowledge of the worst-case execution time (WCET) of a task, a scheduler also requires information on its worst-case energy consumption (WCEC).

Over the last few decades, the real-time systems community has made significant progress towards precisely determining WCETs [4]. Unfortunately, existing tools cannot be directly used for WCEC analysis: For example, one might propose to estimate the WCEC by multiplying the results of a WCET analysis by the average power consumption of the system. However, such a derivation of WCEC bounds from WCET results usually cannot be done in a safe way [5] as due to the complexity of most state-of-the-art processors there is no safe and close correlation between time and energy. In general, the fact that a task has a lower execution time than another does not always mean that it also consumes less energy [6].

WCEC analysis requires knowledge about both the task program (i.e., the source code) as well as the hardware platform; the latter must be provided by energy models comprising information on the maximum energy consumption of each instruction. As a consequence, the problem of determining the WCEC can be solved by identifying the program path that has the highest energy demand. There are mainly two reasons why this is inherently difficult in practice: First, for non-trivial programs examining all possible paths for all possible program inputs is not feasible due to the huge number of paths that need to be considered; WCET analysis suffers from the same problem of path explosion [7], [8]. Second, for many common low-power microcontrollers creating precise energy models is not possible as detailed documentation on the behavior and energy consumption in different power modes is missing. The situation is complicated by the fact that even smallest ARM microcontrollers (Cortex M0+) already support ten distinct power-saving modes [9]. In addition, the actual energy consumption of a microcontroller highly depends on the manufacturing process [10], making it impossible to define precise absolute numbers.

In this paper, we present *0g*[1], a tool that relies on a combination of different techniques to analyze the WCEC of a task. If precise energy models are available for the target hardware, *0g* performs a static analysis solely based on program code. For scenarios in which this is not possible, the tool provides means to extract suitable input parameters from program code using *relative* energy models in order to determine the WCEC by measurement. In contrast to the absolute energy models discussed above, a relative energy model does not contain exact numbers for the energy cost of each instruction but instead consists of estimated, non-dimensional values characterizing the energy consumption of instructions in relation to each other. The key benefit of this approach is that due to their simplicity relative energy models are much easier to obtain than absolute energy models. Nevertheless, relative energy models are sufficient for *0g* to produce precise WCEC estimations, as we show in our evaluation.

Determining the WCEC by measurement transforms the problem of finding the program path with the highest energy consumption into the problem of finding the input values for this path in order for them to be used as measurement parameters. However, as the two problems are related *0g* is able to apply the same code-analysis techniques for both relative and absolute energy models. Thereby, the tool draws on existing knowledge that mainly originated from timing analysis, namely the implicit path enumeration technique [11], symbolic

[1]The name of our WCEC analyzer *0g* refers to our claim for green optimizations (`-Og`) and computing systems without gravity (`0g`).

execution [12], and genetic algorithms [13]; to our knowledge *0g* is the first tool to make use of genetic algorithms for WCEC analysis. As each of the code-analysis techniques has its strengths and weaknesses, prior to executing an analysis, *0g* selects a technique based on the requirements defined by the user which, for example, may include a limit on the duration of the analysis process or a specification whether the tool should provide an over- or an under-approximation of the WCEC.

While relying on only a single technique for each analysis is usually enough to get results, we show that a combination of multiple techniques can lead to significant improvements in quality and speed. For example, by combining implicit path enumeration and genetic algorithms *0g* is able to calculate a metric for the quality of intermediate results, allowing the analysis to be terminated early if a certain minimum quality level is reached. Apart from combining techniques, *0g* ensures an efficient analysis by executing as many steps as possible on server-grade hardware and only resorting to the embedded system to perform measurements in the final step of a WCEC analysis with a relative energy model.

In summary, the main contributions of this paper are:

1) *0g*, a comprehensive approach to WCEC analysis exploiting several code-analysis techniques.
2) A method that enables precise WCEC computations based on relative energy models and measurements.
3) An approach to improve WCEC analysis by combining different techniques.
4) An evaluation of the current *0g* prototype including experiments using three distinct hardware platforms.

The remainder of the paper is structured as follows: Section II presents our view of an energy-aware task scheduler and identifies requirements that arise with regard to WCEC analysis. Section III details the concept of *0g* and describes how the tool combines different techniques for WCEC analysis. Section IV provides information on the implementation of our current prototype and gives details on relative energy models. Section V verifies the feasibility of our approach through an evaluation. Section VI discusses design decisions and possible alternatives. Section VII presents related work, and Section VIII concludes.

II. Energy-Aware Scheduling

Our work targets embedded systems in which scheduling decisions are based on both real-time constraints as well as energy consumption (see Figure 1). This means that a task scheduler, for example, has to take into account (a) whether it can guarantee that a task meets its deadline and has to consider (b) whether there is enough energy available for the task to complete execution. In this section, we present the basic concepts allowing a system to perform energy-aware scheduling. In the remainder of this paper, we thereby mainly concentrate on the energy-related part of the problem.

A. Global System Constraints

Being a scarce resource in the embedded systems we address, there is usually less energy available than necessary to always provide full system functionality. As a result, a task scheduler needs criteria based on which it decides whether

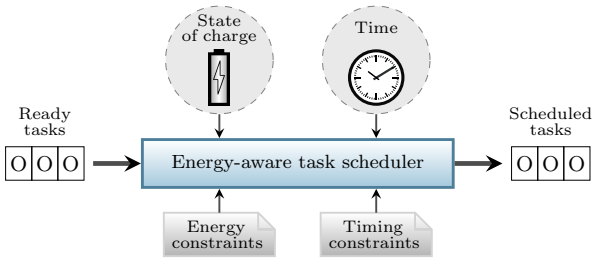

Figure 1: Overview of an energy-aware task scheduler which considers time and energy constraints for scheduling decisions.

or not to schedule a task. As shown in Table I, requirements vary between different system types: If a system is energy limited ① (i.e., the system is initialized with a full battery but is unable to recharge the battery over its lifetime), for example, a common approach is to specify a minimum lifetime during which the system must operate. In such a case, the task scheduler is responsible to guarantee that there is enough energy available to keep essential services running for at least this specific period of time. The same applies to a system that harvests energy ② from the environment in order to achieve a longer lifetime than would be possible with the capacity of its battery. In contrast to the first two types, an energy-neutral system ③ solely depends on harvested energy. As a consequence, using minimum lifetime as a requirement is not useful, as such a system might fail to produce a sufficient amount of energy to reach this threshold in the first place. On the other hand, if a system is able to harvest all the energy necessary to keep essential services alive, its lifetime is basically infinite. In such cases, a possible scheduling requirement could be to provide a certain subset of functionality.

B. Energy Budgets

Energy-aware task scheduling requires the task scheduler to know the amount of energy consumed by a task during execution. As it is inherently difficult for non-trivial tasks to obtain precise numbers, scheduling decisions must be based on estimations of a task's energy consumption. In Section III, we detail our energy-analysis tool that provides such information using a combination of code analysis and measurements. Utilizing the results of the energy analysis, a scheduler allocates a dedicated *energy budget* to each task upon execution. If the actual energy consumption of a task is less than or equal to its energy budget, the task can complete its execution. Otherwise, the task is aborted after having exhausted its given budget, either explicitly by the task scheduler or implicitly due to the system running out of energy. In this paper, we address the goal of minimizing the number of aborted tasks. Consequently, we focus on analyses providing an estimation of the worst-case energy consumption of a task.

System Class	Global System Requirement	One-time Energy Budget	Renewable Energy Buget
① Energy-limited	Min. lifetime	Yes	No
② Energy-harvesting	Min. lifetime	Yes	Yes
③ Energy-neutral	Min. functionality	No	Yes

TABLE I: Although having individual system requirements, all system classes rely on WCEC data for process scheduling.

C. Task Categories: Hard vs. Soft Energy Tasks

Different tasks in a system have different scheduling constraints with regard to energy consumption. Adopting the existing terminology from the domain of real-time systems, we distinguish between two categories: *hard energy tasks* and *soft energy tasks*. A hard energy task may only be executed if the scheduler is able to guarantee that there is enough energy available for the task to run to completion. In particular, we expect this category to include tasks providing essential services (e.g., data-processing task) that must be available for the entire (minimum) lifetime of the system. In contrast, it is safe for the scheduler to abort a *soft energy task* if the execution exceeds the energy budget allocated to the task. Consequently, tasks belonging to this category usually perform work that is not crucial for maintaining system operations (e.g., logging of non-essential status messages). We refer to systems with only soft/hard energy constraints as *soft/hard energy systems* in analogy to soft/hard real-time systems.

The classification into hard and soft energy tasks is orthogonal to the classification into hard and soft real-time tasks, requiring the scheduler to take both domains into account for its decisions. For example, a task with hard energy and hard real-time constraints may only be scheduled if it is guaranteed that both the energy budget as well as the deadline can be met. However, the differences in requirements between hard and soft energy tasks are not only relevant to the scheduler. As we show in Section III, they also allow different techniques to be used for the analysis of a task's energy consumption.

D. Energy Budget Categories: Hard vs. Soft Energy Budgets

As the scheduler ensures that the energy consumption of a task does not exceed its designated energy budget, setting the energy budget of each task to its respective worst-case energy consumption (WCEC) would allow a system to run without ever having to abort a task due to energy-budget violations. Unfortunately, determining the exact WCEC, similar to the WCET, is usually not possible for arbitrary non-trivial tasks as the analysis overhead increases exponentially with task complexity. Therefore, the energy budget of a task must be based on an approximation of its actual WCEC.

We address this problem by distinguishing between two different approaches, one for each task category: As the execution of a hard energy task must always complete, we use a conservatively determined *over-approximation* of the WCEC as *hard energy budget*. In contrast, the fact that a soft energy task may be aborted if it consumes more energy than estimated by the analysis allows us to determine the *soft energy budget* as an *under-approximation* of the actual WCEC.

III. Energy Consumption Analysis

Performing energy-aware scheduling requires knowledge of a task's energy demand during execution. In this section, we present *0g*, a tool that is able to provide such information based on code analysis. To address the particular requirements of each task category, *0g* uses different approaches to determine the energy budgets of hard and soft energy tasks. For this purpose, *0g* combines multiple code-analysis techniques to speed up the analysis and to improve the quality of the results.

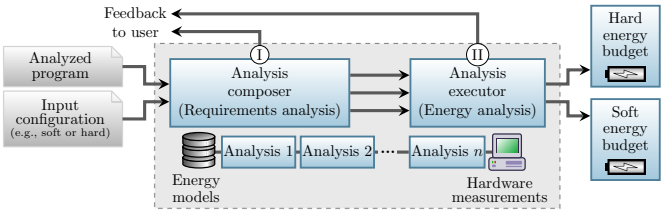

Figure 2: The *0g* WCEC analyzer combines different code-analysis techniques to determine hard and soft energy budgets.

A. Architecture

Figure 2 shows the architecture of the *0g* tool, which performs the energy analysis in two distinct steps: Based on the code of the program to analyze, configuration parameters specified by the user (e.g., task category), and information about the target platform (e.g., energy models, see Section III-B), in the first step (Ⅰ), the analysis composer prepares the analysis to be executed. In particular, this step includes the selection of the analysis techniques to be used (see Sections III-C and III-D) as well as the creation of a schedule specifying how to combine the individual techniques (see Section III-E). Following this, in a second step (Ⅱ), *0g*'s analysis executor performs the actual energy analysis and eventually reports the results back to the user. During each step of the process, the user also receives feedback on the current status and progress of the analysis.

B. Absolute vs. Relative Energy Models

Dimensioning a hard energy budget (see Section II-D) for a task requires detailed information about both the task program as well as the hardware platform it runs on. In this context, especially obtaining the latter is often a problem, as precise energy models comprising absolute numbers for the energy costs of instructions are inherently difficult to create for complex hardware platforms. This is especially true if non-deterministic factors such as cache behavior and pipelining must be taken into account. As a result, hard energy budgets based on the energy models available are usually significant over-approximations of the actual WCECs, especially if they have to assume a cache miss on every memory access.

While the lack of detailed knowledge for the time being makes it necessary to tolerate high over-approximations for hard energy tasks, there is no need to pay the same price for soft energy tasks: Relying on measurements, *0g* is able to provide soft energy budgets that are close under-approximations of tasks' actual WCECs. Note that applying a measurement-based approach transforms the problem of finding the most costly path and computing its energy consumption into the problem of finding the parameters executing this path in order for them to be used as input for measurements. While the former requires precise absolute numbers, *0g* is able to solve the latter based on estimated relative energy costs of instructions, which are easier to obtain and reusable between different but similar target platforms. In Section IV-A, we present details of such relative energy models.

C. Energy Analysis based on Absolute Energy Models

In the following, we discuss how *0g* computes energy budgets for tasks if absolute energy models for the target platform are available. Depending on the category of a task,

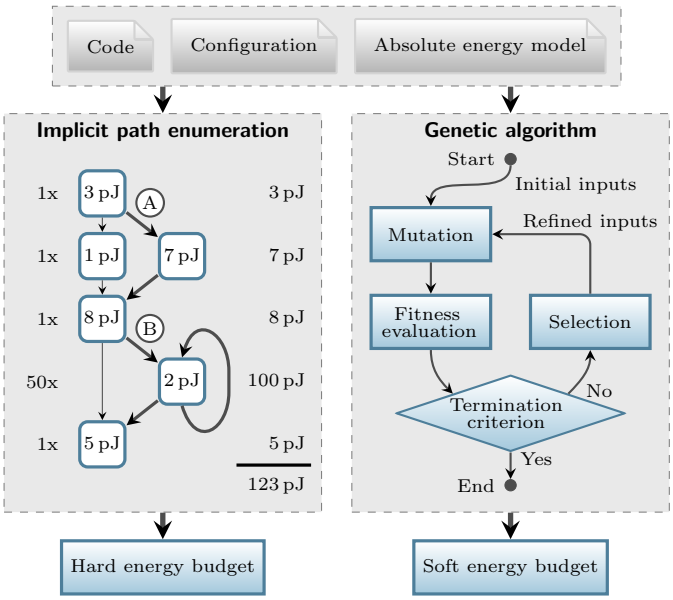

Figure 3: *0g* relies on different techniques for providing hard and soft energy budgets based on absolute energy models.

our tool relies on different techniques (see Figure 3): Hard energy budgets are determined by applying implicit path enumeration [11], whereas genetic algorithms [14] are used for soft energy budgets.

1) Analysis of Hard Energy Budgets: The implicit path enumeration technique (IPET) is a structural approach operating on the control-flow graph of a program, which is modeled through basic blocks (i.e., branch-less sequences of instructions). During the analysis, each basic block first is assigned a specific weight, for example, representing its worst-case energy cost [5]. Then, the flow with the maximum overall weight is identified by solving the corresponding integer linear programming problem.

Exploiting the information contained in an absolute energy model for the target platform, *0g* is able to approximate the WCEC of a basic block by adding up the individual costs of the instructions included in the block; if available in the model, further knowledge about characteristics of the target platform, such as the cache behavior, may also be considered. In combination with the control-flow graph, the WCECs of basic blocks allow *0g* to find the flow consuming the most energy, as shown in the example in Figure 3. Having determined this flow, the tool computes the hard energy budget for the program by multiplying the execution frequency of each basic block in the flow with its respective cost and summing up the values.

There are mainly two reasons why the hard energy budget calculated with the IPET is usually an over-approximation of the actual WCEC: First, the worst-case energy cost of basic blocks by construction are a conservative estimation due to being derived from the worst-case energy costs of instructions as provided by the absolute energy model. Second, being an entirely structural approach, the IPET result may represent an infeasible path that cannot be triggered during execution. For example, it is not guaranteed that all programs matching the control-flow graph in Figure 3 necessarily allow both branches Ⓐ and Ⓑ to be executed subsequently. In Section III-E1, we discuss how to improve the IPET result by relying on a mechanism aimed at detecting such infeasible paths.

2) Analysis of Soft Energy Budgets: Genetic algorithms emulate natural evolution to solve optimization problems. As shown in Figure 3, running a genetic algorithm is an iterative process that is based on two principles: mutation and selection. Starting from an initial set of input values for the analyzed program (in which, for example, all values are set to zero), new sets of input values are generated by introducing variations (i.e., mutation). For each new set, the program is then executed with the corresponding input values in order to determine a *fitness value*. Higher fitness values indicate better solutions and are therefore used to choose the inputs to serve as starting point of the mutation step in the next iteration (i.e., selection). The algorithm terminates if a predefined criterion is met; example criteria include a minimum fitness value, a maximum number of iterations, or an upper bound on the execution time of the algorithm.

A fitness value in *0g* corresponds to the energy consumption of a program for a specific set of input values. Higher values therefore indicate solutions closer to the WCEC. *0g* calculates the fitness value by determining execution frequencies for basic blocks and combining them with information provided by the absolute energy model of the target platform, similar to the approach presented in Section III-C1 to compute hard energy budgets with the IPET. However, in contrast to the IPET, genetic algorithms only consider feasible paths due to actually executing the analyzed program. The longer the analysis runs, the higher the maximum fitness value found by the genetic algorithm. Representing an under-approximation of the WCEC, this fitness value is usable as soft energy budget.

D. Energy Analysis based on Relative Energy Models

Precise absolute energy models are not available for arbitrary target platforms. We therefore present an approach to determine soft energy budgets based on relative energy models comprising estimations on the energy costs of instructions. In contrast to absolute numbers, the relative values used in such models are non-dimensional and thus do not represent actual amounts of energy consumption. As a consequence, it is not possible to directly derive an energy budget from them. However, relative energy models are an effective means to identify input parameters for measuring the energy consumption of a particular program path. Considering only feasible paths, the genetic algorithm usually results in an under-approximation of the WCEC since one cannot ensure that the path leading to the actual WCEC is found within acceptable time. Consequently, this approach is limited to finding soft energy budgets.

Utilizing relative energy models, *0g* analyses the program code using genetic algorithms (see Section III-C2) in order to find a set of input values leading to an execution path as close as possible to the path triggering the WCEC. During this process, the relative energy costs from the model allow *0g* to compute auxiliary fitness values. Although such fitness values cannot directly serve as soft energy budgets, they still enable the genetic algorithm to assess the quality of solutions and therefore to decide which set of input values to select for

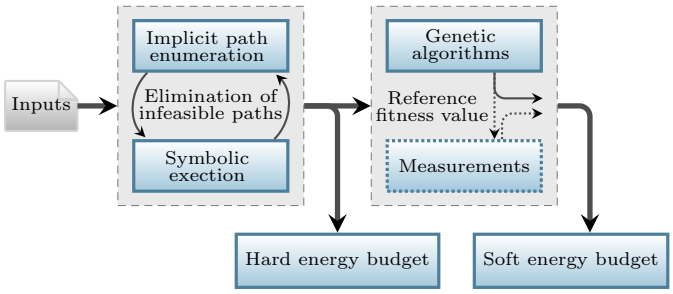

Figure 4: *0g* combines different analysis techniques to improve speed and quality of the analysis.

the next iteration. Note that while this analysis step requires a relative energy model of the target platform, there is no need to actually execute it on the embedded system. Instead, *0g* relies on more powerful systems to speed up the analysis.

With the obtained set of input values from the relative energy model, *0g* in the second step then executes the program on the target hardware and measures its energy consumption. This way, our tool is able to provide an absolute number for the soft energy budget of the corresponding task.

E. Combined Energy Analysis

In Sections III-C and III-D, we have detailed the basic techniques based on which *0g* determines hard and soft energy budgets. Below, we present approaches that allow *0g* to improve the speed and quality of the analysis by combining different techniques (see Figure 4).

1) Improving the Analysis of Hard Energy Budgets: As the IPET is an entirely structural approach, the analysis is not aware of actual program paths (see Section III-C1). Consequently, the hard energy budget calculated with this technique is usually higher than the energy consumed by any feasible path. To mitigate this problem, *0g* applies symbolic execution [12] before delivering a result, in order to detect infeasible paths considered by the IPET [8]. If no indication for an infeasible path is found, the result is returned. Otherwise, *0g* compiles a set of constraints preventing the infeasible path from being considered and restarts the IPET analysis. This way, the amount by which the hard energy budget exceeds the WCEC is reduced in an iterative process.

Detecting infeasible paths on a global scope of the program code is more difficult than on a local scope [15]. Due to the algorithmic complexity preventing a full and thus global symbolic execution of most real-world programs (see Section V-E), the technique can only be applied selectively (i.e., on parts of the code). Therefore, there might be cases in which an infeasible path is not detected. Nevertheless, symbolic execution overall remains an effective means to reduce the over-approximation of hard energy budgets, thereby allowing schedulers to make decisions based on more precise information.

2) Improving the Analysis of Soft Energy Budgets: As discussed in Section III-C2, the execution of a genetic algorithm terminates as soon as a predefined criterion is reached. At this point, the algorithm returns the set of input values that so far has achieved the highest fitness value, which—depending on the energy model—is either an absolute or a relative value. Unfortunately, while a fitness value is a good means to compare two solutions, it contains no information on the global quality of the resulting soft energy budget. That is, even if the algorithm has been executed for a large number of iterations, there is no guarantee that the energy consumption of the best solution found up to this point is necessarily a close under-approximation of the WCEC.

0g addresses this problem by enabling the result of an IPET analysis of the same program to serve as reference value for the quality of a solution. Using this option, as shown in Equation 1, the fitness value fit for a set of inputs is computed as a fraction of the hard energy budget HEB_{IPET} determined by the IPET; f denotes the fitness value as presented in Section III-C2.

$$fit = \frac{f}{HEB_{IPET}} \tag{1}$$

The fitness value fit has a possible range from 0 % to 100 %. Consequently, the search space of the genetic algorithm is bounded through an IPET analysis. However, a fitness value of 100 % is not always reachable, as the hard energy budget is an over-approximation of the WCEC and might be based on an infeasible path; as discussed in Section III-E1, selectively applying symbolic execution mitigates but in some cases does not fully solve this issue. Nevertheless, this does not diminish the approach with regard to cases in which the IPET result corresponds to a feasible path: As we show in our evaluation in Section V-F, utilizing the hard energy budget as reference in such scenarios allows the analysis process to be sped up due to being able to terminate the analysis as soon as a certain minimum fitness value (e.g., 95 %) has been reached.

Calculating fitness values as a fraction of the hard energy budget is not limited to absolute energy models but can also be used to improve the input parameters for measurements (see Section III-D). However, in order to ensure consistency it is important to execute both the IPET as well as the genetic algorithm with the same energy model.

IV. Implementation

We implemented *0g* based on the LLVM [16] compiler framework and its intermediate representation. The modular implementation of our framework seamlessly integrates into the LLVM infrastructure and focuses on reusing analysis results to benefit from combinations of different analysis techniques and energy models. In the following, we discuss the creation and refinement of relative energy models and present details on the current *0g* implementation.

A. Creating and Refining a Relative Energy Model

Obtaining precise absolute energy models is not possible for all hardware platforms, which is why *0g* supports the analysis of soft energy budgets based on relative energy models (see Section III-D). The current relative energy model used by *0g* is implemented on the level of the LLVM intermediate representation and builds on the insight that executing an increased number of instructions leads to higher energy consumption [17]. Like an absolute energy model, the relative model comprises a specific energy cost for each instruction. However, in case of the relative model, this energy cost does not represent an actual energy consumption, but instead is a

non-dimensional number describing the instruction's energy consumption in relation to other instructions. For instructions that lead to the execution of further instructions (i.e., function calls), the list of instruction costs is applied recursively to determine the worst-case energy cost of the called function. By default, our relative energy model assumes equal costs for each instruction and is therefore independent of specific platforms.

Although relative energy models are implemented on a target-independent representation of the code, they can be further refined through target-specific knowledge. For example, if a memory access is known to be expensive on a particular target, its cost is increased in the relative model in comparison to other instructions (e.g., simple arithmetic operations). Equal considerations also apply if floating-point units must be used for the execution of instructions. Furthermore, an additional way to refine the accuracy of a relative energy model is to introduce knowledge about cache behavior; otherwise, the model needs to assume a cache miss on every memory access.

B. Analysis Composition and Execution

As discussed in Section III-A, prior to actually executing the energy analysis, *0g* selects the techniques to be used and decides how to combine them in order to get the best results. While usually combining different techniques yields significant benefits, there are specific cases in which it is advantageous to only rely on a single technique. For example, if the program to analyze does not comprise branches, the tool knows in advance that the IPET result will correspond to a feasible path. As a consequence, *0g* in such cases saves resources by leaving out the detection of infeasible paths with symbolic execution.

When the composition step is complete *0g* starts the analysis, thereby executing as many steps as possible on server-grade hardware and only resorting to the target embedded system for measurements (see Section III-D). That is, most of the analysis is performed on machines with comparably high computing power. To further speed up the process, *0g* is able to parallelize parts of the analysis, for example, by exploiting the parallel nature of genetic algorithms.

C. Implicit Path Enumeration

Our infrastructure for implicit path enumeration is based on the WCET-analysis mechanism utilized in the Real-Time Systems Compiler [18] and is implemented on the level of the LLVM intermediate representation. To determine the upper bound of execution frequencies, the IPET requires upper bounds for loop iterations and recursion depths. For loops, we developed a loop-bound detection mechanism based on chains of recurrences [19], which are used as representation in LLVM's scalar-evolution analysis. This mechanism allows the determination of loop bounds even for nested loops, as shown in our evaluation (see Section V-D1). However, if loop bounds cannot be determined statically, *0g* reports this unboundedness to the user and offers the possibility to provide manual annotations. The same annotation mechanism is also applied for recursion depths. A user of *0g* is able to insert these annotations directly into the code of the analyzed program. The refinement of the IPET through detecting infeasible paths is achieved through the symbolic execution engine KLEE [12].

In the current *0g* implementation, the relative energy model is integrated into the LLVM intermediate representation allowing input-data determination without absolute energy models. In order to integrate an absolute energy model for a specific target platform into the intermediate representation, the mapping to machine code must be known. The LLVM backend stores a mapping between the intermediate representation and assembly when generating machine code [20]. This mapping is exploited to reconstruct the lowering to machine code. Applying an absolute energy-cost model to machine code yields the costs for the IPET together with this mapping.

D. Genetic Algorithms

0g comprises a framework for genetic algorithms that is highly configurable. Amongst other options, for example, the tool allows different strategies to be applied to select the input values for the next mutation step, including stochastic universal sampling [21], tournament selection [22], and best selection (i.e., the best-fitting values are selected). Furthermore, *0g* offers the possibility to start the genetic algorithm with different initial sets of input values (e.g., a set containing random values or a set that only consists of zeros).

We expect a typical use case for *0g* to be scenarios in which a programmer occasionally triggers an energy consumption analysis while still developing the program. As a result, it is likely that the pieces of code analyzed by subsequent runs of the tool share great similarities. To speed up the analysis in such cases, *0g* stores intermediate results of the genetic algorithm (e.g., the best-fitting sets of input values) to disk. This way, a subsequent analysis run does not have to start from scratch but can be initialized with the best solution found up to this point. Furthermore, by also storing the input values for measurements, *0g* facilitates the integration of new target platforms: For example, if the new hardware matches a relative energy model for which the tool has already performed an analysis of the program, the parameters identified by the genetic algorithm can be reused and only the measurement step needs to be executed.

V. Evaluation

In this section, we evaluate the accuracy of *0g* for three target platforms: an ARM Cortex-M0+ (Freescale FRDM-KL46Z [9]), an ARM Cortex-M3 (EFM32 Giant Gecko [23]), and an Intel Core i7-3770 (8 cores, 8 GB RAM). While the first two are typical microcontrollers for embedded systems, the last one allows us to assess the results of *0g* for a complex hardware platform. In addition, we analyze the performance of our tool and different configurations of the genetic algorithm.

A. Experimental Setting

In all our experiments, *0g* performs the code analysis on an AMD Opteron server (48 cores, 64 GB RAM). However, the methods for measuring the energy consumption of a program differ between target platforms: The Intel Core i7 provides the Running Average Power Limit (RAPL) interface [24], which allows fine-grained energy measurements without expensive external measuring hardware. In order to compensate the relatively slow update rate of the employed model-specific registers of about one millisecond, we adopt a strategy proposed

by Hähnel et al. [25] that aligns the start and end of a function with the update. Based on their results, we developed a Linux kernel driver, which also handles the preemptible and multi-threaded environment. Further obstacles for energy measurements with the RAPL interface are discussed in [1]. Although this platform does not target the domain of energy-constrained embedded systems, it serves as example to demonstrate the possibility of measurement-based energy analysis for complex platforms. In contrast to the Intel Core i7, the ARM Cortex-M3 board offers an integrated possibility of current measurements up to an accuracy of 1 µA [26]. For the evaluated program, the sampling rate of 6,250 samples per second is sufficient. For the ARM Cortex-M0+ platform, we use the external energy measurement device that we presented in [6]. The device is based on a current-to-frequency conversion allowing precise energy measurements without sampling rate limitations.

B. Benchmarks

The Mälardalen benchmarks [27] are a widely used evaluation scenario to prove the effectiveness of WCET analysis approaches [8], [15], [20]. In our experiments, we therefore use the *0g* WCEC analyzer to determine energy budgets for several different programs from the Mälardalen benchmark suite. In addition, we analyze an implementation of the bubble-sort algorithm with 2,000 32-bit integer values as input.

The bubble-sort algorithm has several characteristics that make it ideal for our evaluation: First, the benchmark has an input array with static size, which makes it a viable substitute for real-time signal-processing applications. Second, the energy consumption varies significantly for different input values, as the control-flow graph highly depends on the data to process. Third, based on the program code, we are able to manually determine the path leading to the worst-case energy consumption and to assemble a set of input values for it, which is a list sorted in reverse order. This is important as it allows us to verify the correctness of the input values identified by *0g*. Furthermore, for this particular benchmark, we can measure the energy consumption to provide a WCEC and compare it to the energy budgets provided by our tool. In general, detecting the WCEC path is costly and error-prone, creating the need for tools such as *0g*.

C. Hard Energy Budget Analysis

In the first experiment, we use *0g* to determine a hard energy budget for the bubble-sort implementation; the target platform is Cortex-M0+, for which we have developed an absolute, instruction-level energy model. As discussed in Section V-B, we are able to manually reproduce the WCEC of the bubble-sort benchmark in order to have a baseline to evaluate our tool; on the Cortex-M0+ platform this baseline is 100.37 mJ, as shown in Table II. For comparison, the hard energy budget put out by *0g* after the analysis is 119.59 mJ, about 19 % more than the actual WCEC. As the WCEC of a program usually cannot be determined exactly, for a hard energy task it is necessary to make scheduling decisions based on such an over-approximation to ensure that the task runs to completion. In contrast, for soft energy tasks the scheduler may consider under-approximations, as evaluated below.

Analysis Method	Energy E	Normalized $\frac{E}{WCEC}$
Manually (WCEC)	100.37 mJ	100.00 %
0g – Hard energy budget	119.59 mJ	119.15 %
0g – Soft energy budget (95 % fitness)	95.05 mJ	94.70 %

TABLE II: Results of the WCEC analysis of the bubble-sort benchmark for the Cortex-M0+ platform.

Analysis Method	Energy E	Normalized $\frac{E}{WCEC}$
Manually (WCEC)	159.99 mJ	100.00 %
0g – Soft energy budget (95 % fitness)	138.15 mJ	86.35 %

TABLE III: Results of the WCEC analysis of the bubble-sort benchmark for the Cortex-M3 platform.

Analysis Method	Energy E	Normalized $\frac{E}{WCEC}$
Manually (WCEC)	132.10 mJ	100.00 %
0g – Soft energy budget (95 % fitness)	127.66 mJ	96.64 %

TABLE IV: Results of the WCEC analysis of the bubble-sort benchmark for the Intel Core i7 platform.

D. Soft Energy Budget Analysis

In this section, we present experiments evaluating the accuracy of the soft-energy budgets determined by *0g* for the bubble-sort and Mälardalen benchmarks. For this use case, the energy analysis consists of the following steps: Based on the program and the relative energy model described in Section IV-A, *0g* first applies the IPET in combination with symbolic execution (see Section III-E1) to identify a reference value for the subsequent run of the genetic algorithm (see Section III-E2); that is, the IPET result represents a fitness value of 100%. Relying on the genetic algorithm, in the next step, the tool determines a set of measurement parameters. Finally, *0g* executes the program to analyze on the target hardware and measures the energy consumption (see Section III-D).

1) Bubble-Sort Benchmark: For the soft energy budget analysis of the bubble-sort benchmark, we configure *0g* to terminate the search for suitable measurement inputs as soon as the genetic algorithm reaches a minimum fitness value of 95 %. Table II shows the result of the analysis for the Cortex-M0+ platform, which corresponds to the hard energy budget analysis presented in Section V-C. *0g* returns a soft energy budget of 95.05 mJ, which is only 5.3 % less than the actual WCEC. This confirms that *0g*'s approach relying on a relative energy model and measurements is able to provide a close under-approximation of the WCEC.

The third column of Table II shows the results normalized to the actual WCEC. This allows us to assess the quality of the fitness value as an indicator for the accuracy of the soft energy budget determined by *0g*. Here, the soft energy budget represents 94.7 % of the actual WCEC, which is close to the 95 % predicted by the fitness value of the inputs used for the measurement. A key reason for this precise prediction lies in the fact that for the bubble-sort benchmark the IPET result, serving as a reference fitness value, corresponds to a feasible path. As a consequence, a fitness value of 100 % can theoretically be reached. Additionally, for this benchmark our loop-bound detection mechanism based on a scalar-evolution analysis identifies precise bounds even for the nested loop.

Benchmark	Fitness fit	Energy E	Loop Bounds Detected / Total
adpcm	98.79 %	1.36 mJ	14 / 16
bsort100	93.85 %	4.19 mJ	1 / 2
cnt	100.00 %	4.21 mJ	1 / 2
duff	100.00 %	5.27 mJ	2 / 2
edn	100.00 %	6.33 mJ	10 / 11
fir	89.51 %	3.32 mJ	0 / 2
jfdctint	100.00 %	2.19 mJ	2 / 2
st	100.00 %	4.87 mJ	7 / 7

TABLE V: WCEC evaluation of Mälardalen benchmarks

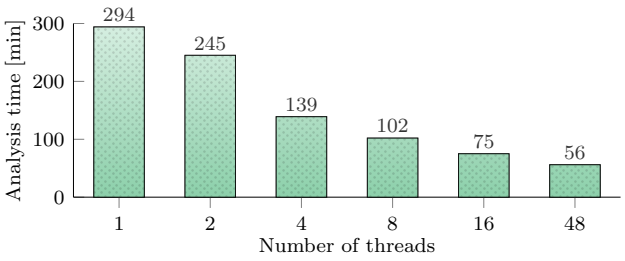

Figure 5: A significant speedup is achieved through parallel execution of the genetic input-determination algorithm.

Performing the energy analysis of the bubble-sort benchmark for the Intel Core i7 as well as the Cortex-M3 platform produces the results presented in Tables IV and III, respectively. For the Intel Core i7, *0g* determines a soft energy budget that is within 3.4 % of the WCEC[2]. For the Cortex-M3 platform, the soft energy budget under-approximates the WCEC by 13.7 %. Although the Cortex-M3 results are not as precise as the results for the other two platforms evaluated, we still consider them acceptable for practical use. More important, the fact that *0g* is able to provide good results for such different target platforms based on the same relative energy model is a significant advantage of our approach.

2) Mälardalen: For our experiments with the Mälardalen benchmark suite on the Intel Core i7, we configure *0g*'s genetic algorithm to terminate if the highest fitness value found is stable for 20,000 consecutive iterations. As shown in Table V, this allows our tool to reach a perfect fitness value of 100 % for five of the eight benchmarks investigated. This is possible because, as for our bubble-sort implementation evaluated in Section V-D1, in these cases the result of the improved IPET analysis corresponds to a feasible path. In contrast, for the other three benchmarks (i.e., `adpcm`, `bsort100`, and `fir`) the IPET provides an actual over-approximation representing an infeasible path, which results in the genetic algorithm terminating with fitness values lower than 100 %.

Besides evaluating the genetic algorithm, the variety of the Mälardalen benchmark suite also allows us to assess the quality of *0g*'s loop-bound detection mechanism for the IPET (see Section IV-C). As presented in Table V, for five of the eight benchmarks *0g* automatically detects bounds for a majority or even all loops in the program. Across all Mälardalen benchmarks evaluated, the detection rate of our tool is 84 %.

E. Performance

In this section, we evaluate *0g* with regard to performance using the bubble-sort benchmark. As shown in Figure 5, on our AMD server the analysis step running a genetic algorithm to identify measurement inputs takes nearly five hours if executed with only a single thread. Allocating eight threads, this time is reduced to 102 minutes; on our Intel Core i7 with eight cores the same procedure requires 104 minutes. Employing all 48 physical cores of the AMD server, a fitness of 95 % is already found after 56 minutes. On the one hand, this confirms the good parallelizability of genetic algorithms. On the other hand, it illustrates the advantage of *0g* being able to execute the most costly parts of the analysis on server-grade hardware.

To put these numbers into perspective, we run an experiment investigating a different approach to find suitable input values for measurements: symbolic execution. Note that due to the prohibitively large computational cost associated, we are not able to perform this experiment with the same input size as for *0g* (i.e., 2,000 integer values), which is why we truncate the input list to nine elements. Despite the reduction by two orders of magnitude, the experiment conducted with the symbolic execution engine KLEE [12] still takes more than five days to complete during which the analysis explores 130,026 program states. For comparison, for an input size of nine, thanks to the genetic algorithm *0g* yields a result with a fitness value of 95 % after two seconds.

F. Genetic Algorithm Configuration

As discussed in Section IV-D, *0g*'s framework for genetic algorithms is highly configurable. In the following, we evaluate the impact of a key parameter, the mutation rate, specifying the degree of variation introduced during the mutation step in each iteration. For a mutation rate of 100 % the genetic algorithm degrades to random testing as the mutation process in one iteration is completely independent of the set of values selected in the previous round. In contrast, a low mutation rate ensures with high probability that the progress made is carried into the next round. Figure 6 shows the evolution of fitness values in the bubble-sort benchmark for two such mutation rates (i.e., 0.005 % and 0.2 %). In both cases, the experiment starts with the initial input values all set to zero, which for this use case results in a fitness value of 76.93 %. The longer the analysis runs, the higher the maximum fitness value found by the genetic algorithm, eventually reaching the termination criterion of 95 %. For a mutation rate of 0.005 % this takes about 18,700 iterations, while for a mutation rate of 0.2 % the analysis already terminates after about 11,600 iterations due to the higher degree of variation enabling a faster progress.

Figure 6 illustrates that there is a general trade-off between accuracy and effort with regard to the analysis of soft-energy budgets: Providing more computing power and/or spending additional time leads to better results. In this context, the fact that the fitness value can serve as a metric for accuracy is of particular importance, as it provides a reference point for the overall progress of the analysis. By specifying a fitness

[2]The complex caching behavior of the Intel Core i7 platform makes it difficult to precisely determine the WCEC by measurement, as it cannot be ensured that, for example, all possible cache misses are considered. We therefore repeated the experiment for this platform with complex hardware features, including caches, disabled. For this scenario, the soft energy budget provided by *0g* represents an under-approximation of about 4%.

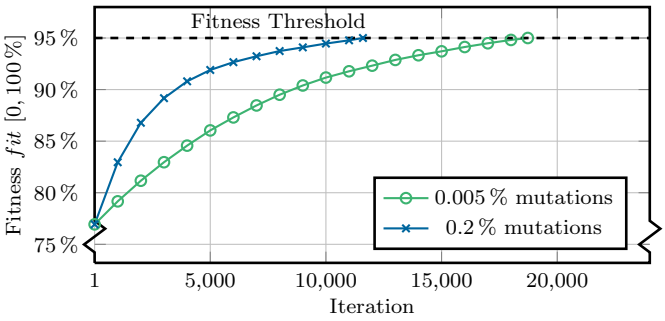

Figure 6: Evaluation of different mutation rates in the genetic algorithm for the bubble-sort benchmark.

threshold as a termination criterion for the genetic algorithm, a *0g* user is able to instruct the tool to end the analysis as soon as a certain accuracy is reached. In many cases, this significantly speeds up the process compared with waiting for the algorithm to complete after a predefined number of iterations.

VI. Discussion

In the following, we focus on a number of design decisions made during the development of *0g* and discuss possible alternatives and extensions.

Choice of Task Categories: The main purpose of differentiating between hard and soft energy tasks in this paper was to illustrate that there are different requirements for different tasks when it comes to using energy consumption as a scheduling criterion. For some tasks an under-approximation of the WCEC is acceptable, while others by all means must run to completion, requiring their energy budgets to be based on an over-approximation of the WCEC. While these two task categories are likely to be sufficient for some embedded systems, we expect to see energy-aware schedulers that consider additional task categories including, for example, hard energy tasks that may be interrupted and suspended in order to be resumed at a later point in time.

Use of Relative Energy Models: If an absolute energy model for the target platform is not available, *0g* can use a relative energy model to obtain input values for measuring soft energy budgets. In principle, it would also be possible to execute the genetic algorithm without any energy model. However, this would require to evaluate the fitness value of program inputs through measurements and consequently to perform a significant part of each iteration on the comparably slow target platform. In contrast, the relative energy model allows *0g* to minimize the analysis time by running the most costly steps on hardware with high computing power.

In our evaluation, we used the same relative energy model to determine soft energy budgets for three different target platforms. *0g* provided acceptable results in all cases. However, for scenarios that require an even higher accuracy, our approach allows to refine the relative energy model by utilizing additional knowledge about a particular target platform. In general, there is a tradeoff between accuracy and reusability: On the one hand, target-specific relative energy models may lead to better results; on the other hand, the measurement inputs identified by the genetic algorithm based on a highly target-specific model usually can no longer be reused for measuring soft energy budgets on other target platforms. In consequence, we expect the cost of refining a relative energy model to be paid only if high accuracy is actually necessary.

Reference Values: The hard and soft energy budget of a task represent an upper and lower bound of the WCEC, respectively. Thus, the distance between the two energy budgets serves as an indicator for the quality of the result returned by *0g*. If the distance is large, the user may want to grant additional resources (e.g., time or processor cores) to the tool in order to intensify the analysis. In contrast, a small distance between hard and soft energy budget proves that each of them is close to the WCEC, offering additional flexibility. For example, at only a small extra cost, a soft energy task can be given the same run-to-completion guarantees as a hard energy task by scheduling both based on their hard energy budgets.

VII. Related Work

We are not the first to argue that schedulers of embedded real-time systems should be made energy aware. Völp et al. [1], for example, even showed that for some mixed-criticality systems rudimentary energy awareness is not enough in order to ensure all real-time guarantees. They conclude that in such cases, when it comes to making scheduling decisions, energy must be treated equally to time. Especially for such systems, precise information on the WCEC of tasks is essential.

Although WCET analyzers usually cannot be directly used for determining hard and soft energy budgets, *0g* is able to benefit from ongoing research and advances made in the field of code analysis in general and WCET analysis in particular. For example, integrating additional approaches to refine the IPET result could lead to more precise hard energy budgets as well as improved reference fitness values for the genetic algorithm. A possible candidate in this context is the technique proposed by Blackham et al. [15] to detect and eliminate infeasible paths through exploring mutually unsatisfiable constraints. Furthermore, we expect *0g* to benefit from ongoing improvements of symbolic execution platforms [28].

Wegener [13] evaluated several strategies for using evolutionary algorithms to verify timing results. For *0g*, we utilize the basic concepts of this work for WCEC analysis. Furthermore, we provide a means to bound the search space of an evolutionary algorithm through an initial IPET analysis.

Wenzel et al. [29] addressed a key problem of measurement-based WCET analysis: the generation of test data. They proposed a systematic approach which first applies techniques with comparably low computational overhead (e.g., reusing the test data from previous analysis runs or setting all input variables to random numbers) and then gradually proceeds to more costly techniques (e.g., heuristics and model checking) if the quality of the test data produced by the initial stages is insufficient. With *0g* relying on measurements to determine soft energy budgets based on relative energy models, its accuracy and efficiency also depends on the ability to quickly obtain good test data. Some mechanisms, such as the reuse of input values for different target platforms, are already integrated into *0g*, but we intend to add further mechanisms in the future.

To the best of our knowledge, only a single WCEC analyzer has been published so far: Jayaseelan et al. [5] presented an approach to provide an over-approximation of the WCEC based on static analysis. In contrast to *0g*, their technique mandates absolute energy models comprising precise information on the maximum energy consumption of each instruction, which are difficult to obtain. Our tool is able to use absolute energy models if they are available. However, if this is not the case, *0g* can also provide precise results based on relative energy models and measurements. Furthermore, our approach combines several techniques, which allows *0g* to tailor the analysis to the specific requirements of the user, for example, by applying different techniques for over-approximations than for under-approximations of the WCEC.

VIII. Conclusion

Energy-aware scheduling requires knowledge about the WCEC of tasks. In this paper, we presented *0g*, an approach to retrieve such information for different target platforms by combining several techniques, including implicit path enumeration and genetic algorithms. If an absolute energy model is available, the tool performs the entire analysis on server-grade hardware. Otherwise, it identifies suitable program inputs using a relative energy model and determines an under-approximation of the WCEC by measurement on the actual target hardware. Our evaluation has shown that such a measurement-based approach can provide precise WCEC estimates.

Acknowledgments

We thank Fabian Scheler, Volkmar Sieh, Sebastian Maier, and Christopher Eibel for the insightful discussions and comments. This work is supported by the German Research Foundation (DFG), in part by Research Grant no. SCHR 603/9-1 (Aspect-Oriented Real-Time Architecture), Research Unit FOR 1508 under grant no. KA 3171/3-1, Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89, Project C1), and the Bavarian Ministry of State for Economics under grant no. 0704/883 25 (EU EFRE funds).

References

[1] M. Völp, M. Hähnel, and A. Lackorzynski, "Has energy surpassed timeliness? – Scheduling energy-constrained mixed-criticality systems," in *Proceedings of the 20th Real-Time and Embedded Technology and Applications Symposium*, 2014, pp. 275–284.

[2] E. Bini, G. Buttazzo, and G. Lipari, "Speed modulation in energy-aware real-time systems," in *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, 2005, pp. 3–10.

[3] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 89–102, 2001.

[4] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem – Overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–53, 2008.

[5] R. Jayaseelan, T. Mitra, and X. Li, "Estimating the worst-case energy consumption of embedded software," in *Proceedings of the 12th Real-Time and Embedded Technology and Applications Symposium*, 2006, pp. 81–90.

[6] T. Hönig, H. Janker, C. Eibel, O. Mihelic, R. Kapitza, and W. Schröder-Preikschat, "Proactive energy-aware programming with PEEK," in *Proceedings of the 2014 Conference on Timely Results in Operating Systems*, 2014, pp. 1–14.

[7] R. Kirner and P. Puschner, "Obstacles in worst-case execution time analysis," in *Proceedings of the 11th Symposium on Object Oriented Real-Time Distributed Computing*, 2008, pp. 333–339.

[8] J. Knoop, L. Kovács, and J. Zwirchmayr, "WCET squeezing: On-demand feasibility refinement for proven precise WCET-bounds," in *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, 2013, pp. 161–170.

[9] Freescale Semiconductor Inc., "KL46 sub-family reference manual," 2013.

[10] P. Hurni, T. Braun, B. Nyffenegger, and A. Hergenroeder, "On the accuracy of software-based energy estimation techniques," in *Proceedings of the 8th European Conference on Wireless Sensor Networks*, 2011, pp. 49–64.

[11] P. Puschner and A. Schedl, "Computing maximum task execution times: A graph-based approach," *Real-Time Systems*, vol. 13, pp. 67–91, 1997.

[12] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, 2008, pp. 209–224.

[13] J. Wegener, "Evolutionary testing of the temporal behaviour of real-time systems," Ph.D. dissertation, Humboldt University of Berlin, 2001.

[14] J. H. Holland, *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.

[15] B. Blackham, M. Liffiton, and G. Heiser, "Trickle: Automated infeasible path detection using all minimal unsatisfiable subsets," in *Proceedings of the 20th Real-Time and Embedded Technology and Applications Symposium*, 2014, pp. 169–178.

[16] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004, pp. 1–12.

[17] J. Pallister, S. Hollis, and J. Bennett, "Identifying compiler options to minimise energy consumption for embedded platforms," *Computing Research Repository, arXiv*, pp. 1–14, 2013.

[18] F. Scheler and W. Schröder-Preikschat, "The RTSC: Leveraging the migration from event-triggered to time-triggered systems," in *Proceedings of the 13th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 2010, pp. 1–8.

[19] O. Bachmann, P. S. Wang, and E. V. Zima, "Chains of Recurrences – a method to expedite the evaluation of closed-form functions," in *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, 1994, pp. 1–8.

[20] B. Huber, D. Prokesch, and P. Puschner, "Combined WCET analysis of bitcode and machine code using control-flow relation graphs," in *Proceedings of the 14th Conference on Languages, Compilers and Tools for Embedded Systems*, 2013, pp. 163–172.

[21] J. E. Baker, "Reducing bias and inefficiency in the selection algorithm," in *Proceedings of the International Conference on Genetic Algorithms and their Applications*, 1987, pp. 14–21.

[22] D. E. Goldberg and K. Deb, "A comparative analysis of selection schemes used in genetic algorithms," *Foundations of Genetic Algorithms*, pp. 69–93, 1991.

[23] Silicon Laboratories Inc., "EFM32GG reference manual," 2013.

[24] Intel Corporation. (2013) Intel 64 and IA-32 architectures software developer's manual: System programming guide, part 2.

[25] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using RAPL," *Performance Evaluation Review*, vol. 40, no. 3, pp. 13–17, 2012.

[26] Silicon Laboratories Inc., "User manual starter kit EFM32GG-STK3700," 2013.

[27] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks: Past, present and future," in *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, 2010, pp. 136–146.

[28] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: Design, implementation, and applications," *ACM Transactions on Computer Systems*, vol. 30, no. 1, pp. 1–49, 2012.

[29] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based timing analysis," *Leveraging Applications of Formal Methods, Verification and Validation*, vol. 17, pp. 430–444, 2008.