Static Worst-Case Analyses and Their Validation Techniques for Safety-Critical Systems

Peter Wägemann

Abstract The reliable operation of systems with both timing and energy requirements is a fundamental challenge in the area of safety-critical embedded systems. In order to provide guarantees for the execution of tasks within given resource budgets, these systems demand bounds of the worst-case execution time (WCET) and the worst-case energy consumption (WCEC). While static WCET analysis techniques are well established in the software development process of real-time systems nowadays, these program analysis techniques are not directly applicable to the fundamentally different behavior of energy consumption and the determination of the WCEC. Besides the missing approaches for WCEC bounds, the domain of worst-case analyses generally faces the problem that the accuracy and validity of reported analysis bounds are unknown: Since the actual worst-case resource consumption of existing benchmark programs cannot be automatically determined, a comprehensive validation of these program analysis tools is not possible.

This summary of my dissertation addresses these problems by first describing a novel program analysis approach for WCEC bounds, which accounts for temporarily power-consuming devices, scheduling with fixed real-time priorities, synchronous task activations, and asynchronous interrupt service routines. Regarding the fundamental problem of validating worst-case tools, this dissertation presents a technique for automatically generating benchmark programs. The generator combines program patterns so that the worst-case resource consumption is available along with the generated benchmark. Knowledge about the actual worst-case resource demand then serves as the baseline for evaluating and validating program analysis tools. The fact the benchmark generator helped to reveal previously undiscovered software bugs in a widespread WCET tool for safety-critical systems underlines the relevance of such a structured testing technique.

P. Wägemann (✉)
Friedrich-Alexander University Erlangen-Nürnberg (FAU), Chair in Distributed Systems and Operating Systems, Erlangen, Germany
e-mail: wagemann@cs.fau.de

M. Felderer et al. (eds.), *Ernst Denert Award for Software Engineering 2020*,
https://doi.org/10.1007/978-3-030-83128-8_11

1 Introduction

One central challenge of the 2020s in the domain of embedded computing systems is the reliable operation of systems that face energy constraints due to harvesting energy from the environment [4]. Additional to the existence of energy constraints, applications increasingly come up with both energy and timing constraints, such as medical devices (like implantable defibrillators) with harvesting mechanisms [16]. Such systems have to meet real-time guarantees on the timely execution of tasks, and additionally, tasks have to be executed within given energy budgets due to being battery-operated and dependent on energy-harvesting techniques. In order to enable safe scheduling under consideration of available time and energy resources, developers require the values of the *worst-case execution time (WCET)* as well as the *worst-case energy consumption (WCEC)* of each task.

Static program analysis tools of the system's program code are the fundamental means in the domain of real-time systems to determine the tasks' bound of the WCET [32]. While practical analyses for the timing-related problem exist for systems with fixed real-time priorities [1, 5], these techniques are not directly applicable to the fundamentally different problem of energy consumption and the assessment of WCEC values. For example, for the WCEC-related problem, analyses that only consider the real-time priorities of tasks are insufficient: Tasks with a low real-time priority have the possibility to temporarily activate power-consuming devices (e.g., transceivers). The activation of a device, in turn, influences the power demand (and likewise the energy demand over time) of each task in the system irrespective of their priority. Thus, worst-case energy-consumption analyses need to consider such *mutual influences* between tasks in the whole system.

Besides the missing approaches for WCEC values, static worst-case analysis tools generally face the fundamental problem of missing evaluations and validations of the reported worst-case bounds: Based on sound abstraction, analysis tools yield safe resource-consumption bounds. However, the actual WCET and likewise the actual WCEC of arbitrary (benchmark) programs are unknown since such nontrivial properties cannot be automatically extracted from existing programs [12, 20]. Due to this missing knowledge, the assessment of the analysis tool's accuracy (i.e., the distance between reported bound and actual worst case) is not possible, and the analysis pessimism remains unknown. Furthermore, the lack of the actual worst case leaves the question unanswered whether the implemented analysis contains software bugs since the actual baseline is not available.

The dissertation [33] addresses the mentioned problems of (1) determining WCEC bounds, (2) validating static worst-case analyzer, and (3) operating energy- and time-constrained embedded systems. Figure 1 illustrates the conceptual structure of the dissertation. The following three solutions are the main contributions of the dissertation:

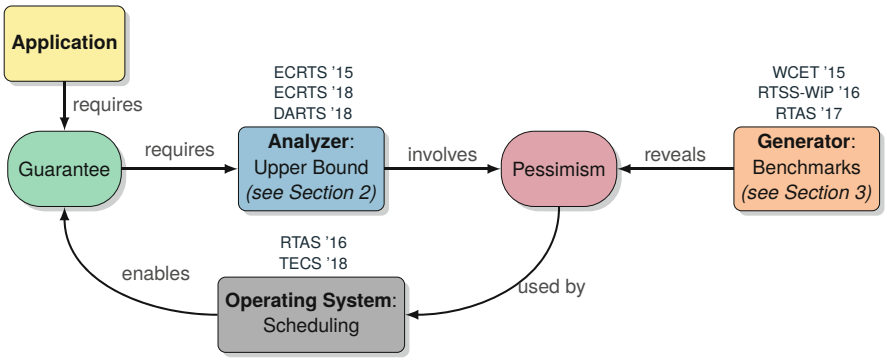

Fig. 1 Conceptual structure of the dissertation: Runtime guarantees demand upper resource-consumption bounds, which are determined by static worst-case tools. A benchmark-generation technique reveals the expected degree of analysis pessimism and thereby validates the reported results of the analysis tools. An operating-system kernel, which makes use of resource bounds, eventually enables applications to operate safely

1. The WCEC analyzer *SysWCEC* determines upper bounds of the energy consumption of tasks in systems that are scheduled with fixed priorities [24, 25, 27] (see Sect. 2).
2. The benchmark generator *GenE* allows for the first time comprehensive evaluations and validations of static worst-case tools based on automatically generated benchmarks, whose actual worst case is known [26, 28, 31] (see Sect. 3).
3. The operating-system kernel *EnOS* supports the reliable operation of systems with both timing and energy constraints based on a priori knowledge of WCET and WCEC bounds [29, 30].

While this chapter gives insight into the first two main contributions with the focus on system-software engineering, the third aspect goes beyond the scope of this chapter. The chapter is structured as follows: Sect. 2 first presents background information on the problem of WCEC analyses and gives insight into the analyzer *SysWCEC*. The fundamental problem of validating worst-case tools is discussed in Sect. 3, along with the solution of *GenE* for this problem. Sect. 4 concludes this chapter.

2 Worst-Case Analyses

The following Sect. 2.1 presents background information on static worst-case analyses and the system model of the dissertation. The main problem statement for WCEC analyses is discussed in Sect. 2.2. Section 2.3 introduces *SysWCEC*, an analyzer for system-wide WCEC analyses.

2.1 Background and System Model

Besides the analysis results' validity, analysis pessimism is a core problem of static worst-case analyses. The causes of this problem are outlined as follows.

2.1.1 Analysis Pessimism

Figure 2 depicts the resource consumption (either energy or execution time in this scenario) for an example task. The resource consumption of the task varies, for example, with different input data or different initial hardware states when starting to execute the task. Static worst-case tools rely on pessimistic assumptions during their analyses that eventually yield safe, however, overestimating bounds. The distance between the actual worst case and the reported upper bound describes the analyzer's pessimism for a safe execution of the task. Static worst-case analysis techniques are subdivided into the phase of (1) program-flow analysis and (2) hardware analysis. The first phase, which also known as path analysis, explores the possible program paths, determines value constraints of variables, and assigns loop bounds in order to yield upper bounds on executed paths. For example, a mediocre path analysis without the ability to precisely model path constraints is forced to pessimistically include all branches with the highest resource demand, irrespective of whether the combination of these branches is an actually feasible program path. The second phase of hardware analysis uses the executable machine code of the application and determines (under consideration of the target's caching and pipelining behavior) the actual cost (e.g., in mJ for WCEC or in ms for WCET) of each executed basic block. Both the path- and hardware-analysis phase contribute to the overall pessimism of this analysis. Section 3 gives further details on how to isolate and assess the factors that contribute to an analyzer's pessimism. Both analyses phases model the runtime behavior of the executed program on the target platform; thus, analyses demand detailed knowledge about the targeted system model, which is outlined as follows.

Fig. 2 Histogram of an example task's resource consumptions

2.1.2 System Model

All executed code of the application is available for the analysis. As usual for embedded systems, the number of tasks is statically known. Due to the scenario of resource-constrained embedded systems, the systems execute the (single) application on one processing core as part of a microcontroller unit. In contrast to desktop machines, these processing cores have limited complexity, which is a beneficial property for static worst-case analyses and allows determining precise hardware models. All tasks in the system are handled by the scheduler according to their fixed priorities. The tasks have the possibility to acquire operating-system resources (e.g., mutexes) for synchronization purposes. These resources as well as their associated users are statically known. Asynchronous interrupts are common in the targeted systems, for example, in the case of timer interrupts for deadline monitoring. For static worst-case analyses, these interrupts pose a considerable challenge since analysis techniques have to encompass the possible dynamic behavior by static means. A requirement for bounding the occurrence of interrupts is that their arrival is bounded by a minimum inter-arrival time (i.e., the minimum timespan between two successive interrupts). For the power consumption of the system (and thus the energy demand over time), software-controlled devices play an essential role. Devices are not necessarily external to the microcontroller unit, such as transceiver devices for communication. Instead, numerous internal devices, such as timer subsystems or analog-to-digital converters, are usually available. Both internal and external devices have the same power-consumption behavior in view of a static analysis tool: The activation of a device leads to an increase of the whole system's power demand and the deactivation, in turn, reduces the power, which is further discussed as follows.

2.2 Problem Statement of WCEC Analysis

For the purpose of energy savings, devices are kept active only for the duration while their service is required. Figure 3 illustrates such a temporary device activation, for example, for sending out a packet via a transceiver device. This temporary device activation is executed by the task τ_{LOW} with low priority in contrast to the second task with higher priority τ_{HIGH}. The task τ_{HIGH} is, in turn, activated through an interrupt service routine (ISR). ISRs generally preempt running tasks, regardless of the tasks' real-time priority. In the example, the low-priority task's device activation leads to an increase of the systems power demand from 5 mW to 35 mW. As illustrated on the right part of Fig. 3, two possible runtime scenarios exist in view of the energy demand from the start until the completion of τ_{LOW}: In scenario a (see $\lightning_{\mathsf{a}}$ in Fig. 3), the interrupt occurs in the system state of lower power demand (i.e.,5 mW). In contrast, the ISR is serviced in scenario b ($\lightning_{\mathsf{b}}$) in the state of the higher power demand. This second scenario is the worst case in terms of the energy demand (i.e., area under the power demand). This worst-case scenario

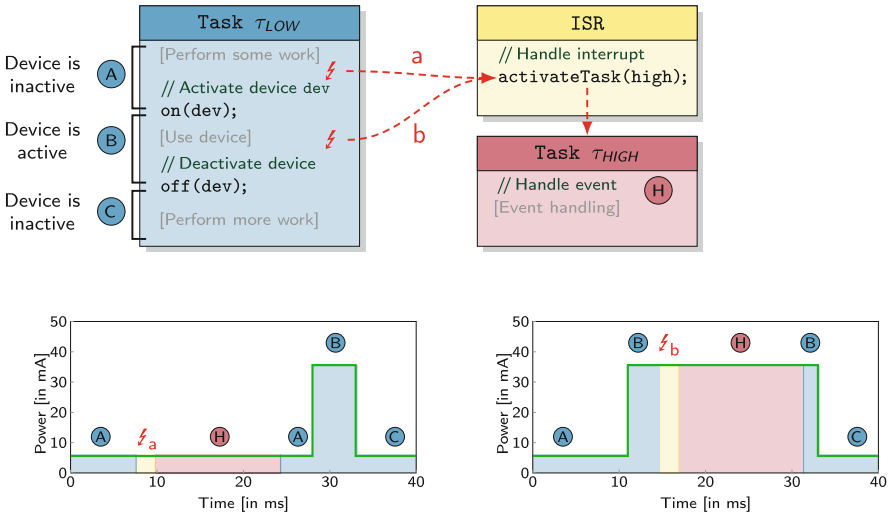

Fig. 3 The system-state–dependent power demand of the whole system influences the energy demand of each task

is indeed the worst case for both τ_{LOW} (whole area) as well as for τ_{HIGH} (red area below (H)). The worst case of τ_{HIGH} is initiated by the fact that the task starts executing with a higher initial power demand (caused by τ_{LOW}). For the analysis of the WCET, these context-sensitive power states are irrelevant, and only the unilateral influence of lower- tasks by higher-priority tasks is of interest. In contrast, WCEC analyses must account for system-wide de-/activations of devices as well as the system's total power state. When reconsidering the example, τ_{HIGH} influences the energy demand of τ_{LOW} by a prolonged execution time. However, also τ_{LOW} influences τ_{HIGH} due to τ_{LOW}'s responsibility for τ_{HIGH}'s higher power state. To summarize the observations: WCEC analyses have to consider all device de-/activations along the system-wide program paths in order to account for the mutual influences between all tasks in the system.

2.3 *SysWCEC: Whole-System WCEC Analysis*

The dissertation proposes the *SysWCEC* approach [25] for solving the problems mentioned above. In a nutshell, *SysWCEC* consists of four steps:

1. *Decomposition*: The code of the application is decomposed into blocks with a common set of active devices and, thereby, a common power state.
2. *Path Exploration*: Based on the result of the decomposition, *SysWCEC* conducts an explicit enumeration of all possible system-wide program paths.

3. *Problem Formulation*: The knowledge about the program paths is the foundation for the formulation of an integer linear program (ILP).
4. *WCEC Determination*: Solving the ILP, with its objective of the worst-case behavior, eventually yields a WCEC bound of the analyzed task.

The following section gives further insight into *SysWCEC*'s working principle, based on the example illustrated in Fig. 3.

2.3.1 Decomposition: Power Atomic Basic Blocks

The left part of Fig. 4 illustrates the original system with the three components of τ_{LOW}, τ_{HIGH}, and the ISR. The step of the decomposition relies on the existing technique of *atomic basic blocks (ABBs)* [21, 22]. The core concept of ABBs is to split up the application's code at a system-call location (e.g., task activation, acquisition of mutex). In other words, each system call forms a terminator in the atomic–basic-block graph. Each ABB thereby forms an atomically schedulable unit from the perspective of the system's real-time scheduler, whereas each ABB may be executed inside a different system state. The *SysWCEC* approach extends this notion of atomic basic blocks to support power-aware considerations. That is, *SysWCEC* uses the device-related system calls (i.e., the calls that change the system's active device configuration) as additional terminators, which results in the graph of *power atomic basic blocks*, or *PABBs* for short. As a result, the PABB graph holds blocks with a common set of active devices and, thus, a common state of the power demand. When considering the result of this decomposition in the running example (see right

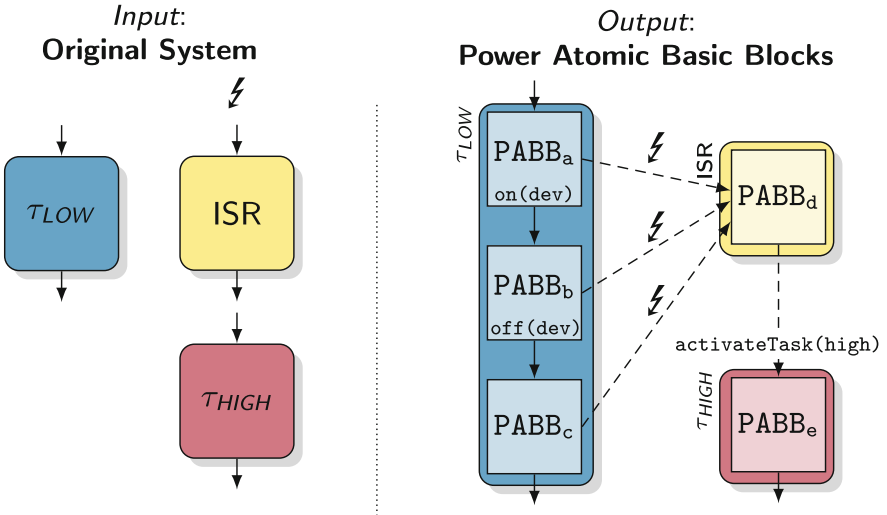

Fig. 4 Decomposition into PABB Graph: The original system is decomposed into blocks with a common set of active devices

part in Fig. 4), the low-priority task τ_{LOW} now consists of three parts: the $\texttt{PABB}_\texttt{a}$ with the code prior to the device activation, $\texttt{PABB}_\texttt{b}$ with the code with the active device, and $\texttt{PABB}_\texttt{c}$ with the code after the device deactivation.

2.3.2 Path Exploration: Power-State–Transition Graph

The subsequent step after the decomposition of the original system into the PABB graph is the path exploration. The input of this analysis step is the PABB graph, and the output is a power-state–aware graph, named *power-state–transition graph (PSTG)*. Figure 5 illustrates the working principle of this path exploration based on the running example: The algorithm starts with the initial power state of the lower power consumption of 5 mW. The path-exploration algorithm accounts for the operating-system semantics (e.g., the fixed-priority scheduling strategy) and the potentially occurring interrupts. For each possible state transition, the PSTG inserts an edge to a corresponding PSTG node that describes the system state. Regarding the handling of varying power demands, the algorithm accounts for device-state changes and for the associated power-demand changes. If no power-state change happens alongside a transition, the path exploration propagates the current power state. The algorithm terminates when all possible states are visited, which is possible due to the bounded number of system states in embedded systems with their fixed number of tasks. As illustrated on the right side of Fig. 5, the PSTG finally holds all context-sensitive program paths with their power consumptions. The knowledge

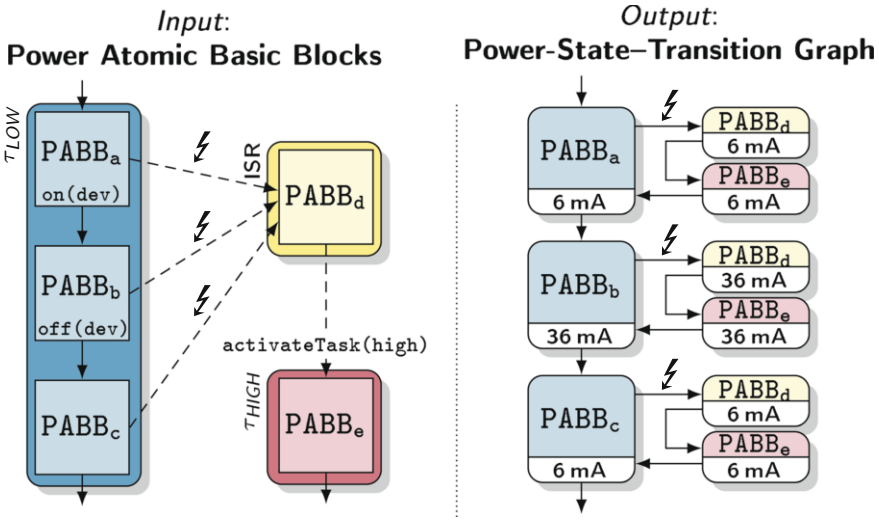

Fig. 5 Path Exploration: The *SysWCEC* approach uses the decomposed system (i.e., PABB graph) and conducts an explicit enumeration of all possible system paths, which results in the power-state–transition graph

about these possible system-wide program paths enables *SysWCEC* to tackle the problem of mutual influences between tasks.

2.3.3 ILP Formulation

The PSTG is the main input for the third analysis step in *SysWCEC*, the formulation of an integer linear program. *SysWCEC* exploits the main technique for formulating maximum flow problems from the well-established approach of the *implicit path enumeration technique (IPET)* [14]. The IPET targets single-threaded executions and uses the program's control-flow graph as input. Based on the control-flow graph's branches, the IPET inserts constraints from the program's flow into an ILP formulation. With the same purpose of finding an upper bound of the cost through a flow graph, the *SysWCEC* approach leverages the IPET's single-threaded approach to the system-state level. *SysWCEC* now uses the PSTG's paths as constraints for an ILP formulation. The objective function of the ILP is shown in the following Eq. 1, which determines the maximum flow through the system's state graph of the analyzed task with its nodes ($v \in \mathcal{V}$) and transitions in between ($\varepsilon \in \mathcal{E}$):

$$max\Big(\underbrace{(\sum_{v \in \mathcal{V}} WCEC(v) \cdot f(v))}_{\text{nodes}} + \underbrace{(\sum_{\varepsilon \in \mathcal{E}} WCEC(\varepsilon) \cdot f(\varepsilon))}_{\text{edges}}\Big) \tag{1}$$

The variables $f(v)$ and $f(\varepsilon)$ denote the execution frequencies of the corresponding nodes and edges. Finally, *SysWCEC* directs this problem formulation to an optimizing solver (e.g., `gurobi`, `lp_solve`), which determines bounds on these execution frequencies and yields the final WCEC bound for the analyzed task.

2.3.4 Cost Modeling

As shown in Eq. 1, the *SysWCEC* approach demands for the context-sensitive costs of each node and each edge in the PSTG. Edge costs are often known from documentation, such as the time and energy demand for the activation of an analog-to-digital converter or transceiver. For the nodes $\mathcal{V}$, *SysWCEC* extracts the costs from the executed code within the PSTG node. For this cost modeling of executed code, *SysWCEC* benefits from a synergy between WCET and WCEC analysis: The multiplication of the maximum (context-sensitive) power demand, which is available for each PSTG node, with the worst-case execution time of the respective node results in an energy-consumption bound for the node. That is, the cost modeling relies on the subsequent Eq. 2:

$$WCEC(v) = P_{max}(v) \cdot WCET(v) \tag{2}$$

That way, the $WCEC(\nu)$ value is *indirectly* determined by means of the maximum power, instead of a direct instruction-level energy-consumption modeling technique [17, 23]. For the determination of $WCET(\nu)$, existing timing-analysis approaches are applied, which also account for the microarchitectural temporal behavior (i.e., caching and pipelining behavior) on the target machine [19].

By employing these techniques, *SysWCEC* determines upper bounds on the energy consumption of a task while considering all other tasks and power-related activities. However, the question on the accuracy of these resource-consumption estimates, from the view of the actual worst case, is left unanswered. The following Sect. 3 focuses on the problem of the analysis accuracy of $WCET(\nu)$ values, which, in turn, contribute to the overall energy-related analysis pessimism according to Eq. 2.

3 Validation of Worst-Case Analyses

The following Sect. 3.1 outlines the main problems of assessing the accuracy of static worst-case analysis tools. One solution for these problems is *GenE*, which is presented in Sect. 3.2.

3.1 Problem Statement of Validating Worst-Case Analyses

As previously illustrated in Fig. 2, the bound reported by the worst-case analysis tool overestimates the actual worst-case resource consumption of the analyzed task. This overestimating scenario assumes a bug-free implementation of the analysis algorithm, which employs abstractions that make pessimistic assumptions on the dynamic runtime behavior. Existing approaches for assessing the degree of analysis pessimism use benchmark suites, which are written on the level of source code (e.g., C). The fundamental problem with this type of evaluation is that the actual baseline is missing, which is the actual worst case serving as ground truth. This problem of missing baselines also prevails when trying to validate if the reported bound actually overestimates the actual worst case.

Unfortunately, the actual worst case as well as all relevant program facts, such as loop bounds or mutually exclusive program paths, cannot be automatically extracted from existing benchmark programs [12, 20]. Moreover, the manual extraction of these program facts is labor-intensive and error-prone. Due to the lack of knowledge on the actual baseline, existing evaluation and validation techniques have limited significance since the absolute degree of over- or—in the presence of software bugs—underestimation on a global scale is unknown.

A further problem when conducting evaluations based on existing benchmark suites is that these programs usually consist of numerous challenges for the static

analyzer. The task $\tau_{\texttt{sort}}$ in the following Listing 1 serves as an example to illustrate this problem. This task's code sorts the numbers held in the array `values`.

Listing 1 Listing with several program components that cause the analysis to potentially overestimate the actual worst-case resource consumption

```
1  const size_t N = 1024;
2  int32_t values[N];
3
4  TASK(τsort){
5    ...
6    for(i = 0; i < N-1; i++){
7      for(j = 0; j < N-1-i; j++){
8        if(compare(values[j], values[j+1]))
9          swap(&values[j], &values[j+1]);
10
11     }
12   }
13 }
```

The outer loop in Line 6 has a constant iteration bound, while the inner loop's bound in Line 7 is decremented with each iteration of the outer loop. Thereby, this nested loop has a rectangular loop shape (i.e., $\texttt{N-1}\cdot\texttt{N-1}$). If analyzers are not able to precisely bound this type of loop, a pessimistic assumption is a rectangular loop shape, which overestimates the actual worst case. Furthermore, input-dependent computations (as outlined in Line 8) cause overestimations if the analyzer cannot automatically determine value constraints. Eventually, the task $\tau_{\texttt{sort}}$ is executed on a hardware platform. Thus, the analyzer has to model the dynamic hardware behavior (i.e., pipelining, caching) in order to report an accurate bound. All these factors contribute to the overall analysis pessimism. When having a poor analysis result, system developers face the problem that the individual causes for the high degree of overestimation are unknown.

3.2 *GenE: Benchmark Generator for WCET Tools*

The *GenE* benchmark [26, 28] is one possible solution for the problem of conducting comprehensive evaluations and validations of worst-case tools. *GenE* generates benchmarks in a way that all program facts are known. Based on this knowledge about these facts, *GenE* is able to determine the actual WCET, which, in turn, serves as ground truth for the validation of static analyzers.

The basic principle of *GenE* is explained best by using a metaphor: Benchmarks are like mazes for analyzers, which need to find a way possible through the maze. However, even if an analyzer finds a way (i.e., solution), it is still unknown if this way is the optimal path. *GenE* follows a different approach: First, *GenE* predefines a path and then successively inserts branches around this path. Thereby, *GenE* builds the maze around this path. Due to *GenE*'s generative approach, the optimal (i.e., actual worst-case) path is known by construction.

3.2.1 Program Pattern

For the process of benchmark generation, *GenE* relies on numerous *program patterns*. These patterns are implemented inside the generator and have the following properties and objectives:

- *Worst-Case–Aware*: The patterns have awareness of their worst-case path and all relevant program facts, such as possible value constraints on introduced variables.
- *Composable*: Patterns have so-called *insertion points* that offer the possibility to insert further program patterns in order to generate new, complex benchmarks.
- *Realistic*: Although *GenE* produces synthetic benchmarks, it aims to output realistic benchmark scenarios that pose realistic challenges for worst-case tools. To solve this problem, *GenE* uses patterns from existing WCET benchmarking suites [8, 9]. Other patterns originate from industry applications or from patterns that are documented in literature to be challenging [3].
- *Resilient*: Compilers have the possibility to decisively change the programs' structure when conducting aggressive optimizations. In order to account for such optimizations, *GenE* uses patterns that already resemble optimized code.

GenE implements these patterns on the level of the LLVM intermediate representation [13]. However, for the sake of readability, the following Listing 2 illustrates a pattern of *GenE* using the C programming language. This pattern, named `init-once`, mimics a lazy initialization of components, which is often found in embedded systems to initialize hardware components.

Listing 2 Pseudo code of *GenE* pattern with insertion points. The actual implementation of program patterns relies on a lower abstraction level (i.e., LLVM intermediate representation).

```
1  static bool initialized = false;
2  void use_hardware(){
3    if (!initialized){
4      // init hardware
5      init(); // insertion point I_1
6      initialized = true;
7    }
8    // use hardware:
9    ... // insertion point I_2
10 }
```

The worst-case analyzer faces the challenge to model the global variable `initialized`. If the analysis is not able to handle such value constraints, it has to pessimistically include the (expensive) call of the function `init()`, which, in turn, causes an overestimating WCET estimate.

Line 5 and 9 in this pattern highlight the two insertion points $\mathcal{I}_1$ and $\mathcal{I}_2$. At these points in the benchmark, further possible patterns are inserted, which are summarized in the following enumeration:

- *Atomic Patterns*: This class of patterns includes arithmetic operations and assignments of constant or computed values.

- *Loop Patterns*: *GenE* implements several shapes of loops, such as nested loops or loops with an input-dependent/constant iteration count.
- *Path Patterns*: The pattern of Listing 2 is part of the path-pattern class. Additionally, this class contains patterns with mutually exclusive paths due to the value constraints or infeasible paths (i.e., dead code).

The fact that *GenE* implements these patterns on the low abstraction level of LLVM intermediate representation gives *GenE* reasonable control over the generated code on the machine-code level without the need to implement patterns directly with a target-specific assembly language. For the mapping between the machine-code representation and the LLVM representation, *GenE* relies on the technique of control-flow–relation graphs [10], which are implemented in the `Platin` toolkit for static analyses [18].

3.2.2 Pattern Suites

In order to tackle the problem of monolithic benchmarks, *GenE* has the notion of *pattern suites*. These suites consist of a subset of all available patterns in *GenE*'s pattern library. For example, the pattern suite `hardware analysis` voids the influence of overestimations due to challenging loops or path constraints. Specifically, *GenE* produces here a single program path with the available patterns (i.e., introduction of variables, arithmetic operations) in the benchmark. This benchmark then challenges the analyzers' ability to model the target's hardware behavior (i.e., caching, pipelining). Regarding the analysis stage of value analysis, *GenE* supports a dedicated suite that inserts variables, arithmetic operations on these variables, and branches based on the value constraints of the computed variables. This suite targets the analyzers' performance in view of the value-range–modeling problem. The main benefit of the pattern suites is the possibility to have benchmarks that are tailored toward a specific scenario (i.e., hardware or path analysis). These scenarios then help developers to reveal individual strengths and weaknesses of analyzers.

3.2.3 Inputs and Outputs of *GenE*

The configuration of the pattern-suite type is one input to the *GenE* generator, as shown in Fig. 6. Besides the suites, *GenE* demands a *path budget*, which approximates the number of instructions (on level of the LLVM intermediate representation) along the generated worst-case path. Increasing the value of the path budget leads to an increase of the benchmark's complexity since a higher budget allows *GenE* to insert more and longer patterns. A further input is the value for the worst-case input value. This value is especially important for the generator because using this value as input for the generated benchmark leads to an execution of the designated worst-case path. This worst-case input value is an integer and also fulfills the purpose of the generator's seed value. That is, varying the worst-case input value

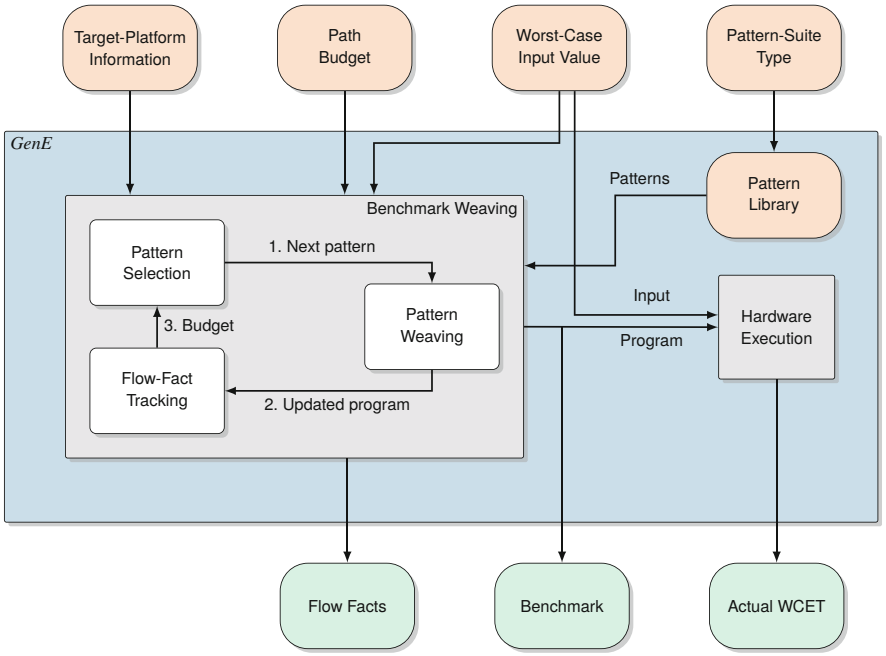

Fig. 6 *GenE* generates benchmarks such that the actual WCET is available along with the generated benchmark. This actual WCET value is necessary for comprehensive evaluations and validations of static analysis tools

also changes the selection of the patterns in a pseudo-random way. By measuring the execution of the worst-case path (either on the target platform or by means of a cycle-accurate simulator), *GenE* determines the actual WCET for the generated benchmark. Thus, *GenE* requires information about the selected target-hardware platform. Besides this actual WCET value, *GenE* provides the generated program along with the relevant flow facts (e.g., loop bounds).

3.3 Benchmark Weaving

As illustrated in Fig. 6, the benchmark-weaving algorithm selects a pattern from the available subset of *GenE*'s pattern library. After inserting the next pattern into the benchmark under construction, the algorithm updates the related flow facts.

An important aspect of the benchmark-weaving algorithm is the mechanism for guaranteeing that the designated worst-case path is—in any case—longer than other (non–worst-case) program paths through the benchmark. Specifically, *GenE* uses a substantially larger path budget for the worst-case path when inserting a branch in

the control flow. For example, along the worst-case path, *GenE* uses a factor of 25 more instructions compared to any other non-worst-case path. With that approach of overweighting the branches of the worst-case path, *GenE* also compensates for varying instruction times due to the target's caching and pipelining behavior. *GenE* supports the configuration of this overweighting factor as a target-specific parameter.

3.4 *Metrics*^{WCA}: Validation of GenE's Benchmarks

The generation of synthetic benchmarks brings up the question of whether the benchmarks are realistic and comparable, for example, with existing benchmark suites in the context of WCET-analysis research. In order to assess the complexity of *GenE*'s benchmarks with other benchmarks, the dissertation presents *Metrics*^{WCA} [31], code metrics for worst-case analyses. *Metrics*^{WCA} relies on several existing complexity measures ($\mathcal{CM}$), which have partly been used in the context of WCET analysis [9]. Some examples of these complexity measures are the number of loops (including their nesting depths) $\mathcal{CM}_{loops}$, the depth of the longest call chain $\mathcal{CM}_{call}$, or McCabe's cyclomatic complexity $\mathcal{CM}_{cc}$ [15]. A novelty of *Metrics*^{WCA} is their property of accounting for the impact of compiler optimizations on the benchmark under evaluation. That is, *Metrics*^{WCA} compare the complexity measures of an optimized version of the benchmark with its original (unoptimized) variant. The result of this comparison is a *resilience factor* $\mathcal{R}$ for a specific complexity measure $\mathcal{CM}$:

$$\mathcal{R}_{\mathcal{CM}} = \frac{\mathcal{CM}_{\text{after optimization}}}{\mathcal{CM}_{\text{before optimization}}} \tag{3}$$

As an example, a benchmark has 12 loops ($\mathcal{CM}_{loops} = 12$) and all loops are optimized out due to the compiler's loop-unrolling optimization (i.e., $\mathcal{CM}_{loops} = 0$ in the optimized variant). As a consequence, this benchmark has a resilience of $\mathcal{R}_{loops} = 0\%$ against the loop-unrolling optimization. From the perspective of assessing an analyzer's performance for determining loop bounds, this benchmark with zero resilience is unsuited since the problem of loop bounds is already straightforward to solve for optimizing compilers. The main observations [26] when applying *Metrics*^{WCA} to *GenE*'s benchmarks are (1) a high resilience against compiler optimizations and (2) comparable complexities with respect to a benchmark suite for WCET analysis [8]. These experiments used the standard configuration of *GenE*'s path budget. A benefit of *GenE*'s generative approach is that an increase of this configuration value leads to benchmarks with larger complexity measures. Thus, the automatically generated benchmark's complexity can be tuned in contrast to existing benchmarks.

Table 1 *GenE* detects individual strengths and weaknesses of the tools' loop-bound analyses

	`constant loop`	`input-dependent loop`	`down-sampling loop`	`triangular loop`
Analyzer `aiT`	✓	✓	✓	✗
Analyzer `Platin`	✓	✓	✗	✓

3.5 *Determining Individual Strengths and Weaknesses of Analyzers with GenE*

GenE's notion of pattern suites enables developers to identify the individual strengths and weaknesses of analyzers. Table 1 shows results of *GenE*'s loop-related suites. The symbol ✓ expresses a successful solution to the respective loop-bound challenge and ✗ indicates that the analyzer reported unbounded loops.

The `constant loop` suite inserts the patterns of variables, arithmetic expressions, and loops with a constant iteration bound. Both WCET analyzers `Platin` [18] and `aiT` [1] can solve this challenge. In the `input-dependent loop` suite, the iteration bound is computed based on the benchmark's input value, and both analyzers find value constraints to bound these loops. The `down-sampling loop` is a loop that decrements the iteration variable in the loop's body in addition to the loop's header. That way, the iteration variable can no longer be described as a closed-form expression. `aiT` solves this challenge while `Platin` fails. The situation is inverted in the `triangular loop` suite: `Platin` internally makes use of the LLVM compiler infrastructure, which supports scalar-evolution expressions [2]. Due to the scalar-evolution analysis, `Platin` is able to solve the challenge of the `triangular loop` suite.

3.6 *Validation of the `aiT` WCET Analyzer*

With knowledge of the ground truth, the actual WCET, *GenE* evaluates the overestimation of analyzers and likewise validates that the reported estimate indeed bounds the worst case. In order to conduct such evaluations, *GenE* generated 10,000 benchmarks for an ARM Cortex-M4 platform (Infineon XMC4500) and compared the actual WCET with the value reported by the commercial analyzer `aiT`. Figure 7 shows a histogram of the occurrences of the overestimations. In these experiments, the geometric mean of all overestimations is 23%. An overestimation of 0% would indicate that all reported bound are equal to the respective actual worst case. However, these experiments also revealed reported values where `aiT` erroneously underestimated the actual WCET. Based on these benchmarks, AbsInt, the company behind `aiT`, could confirm these underestimations. According to AbsInt, these bugs were caused by an erroneous hardware model for memory accesses and the pipelining behavior. Subsequently, AbsInt released a revised version of `aiT`, where

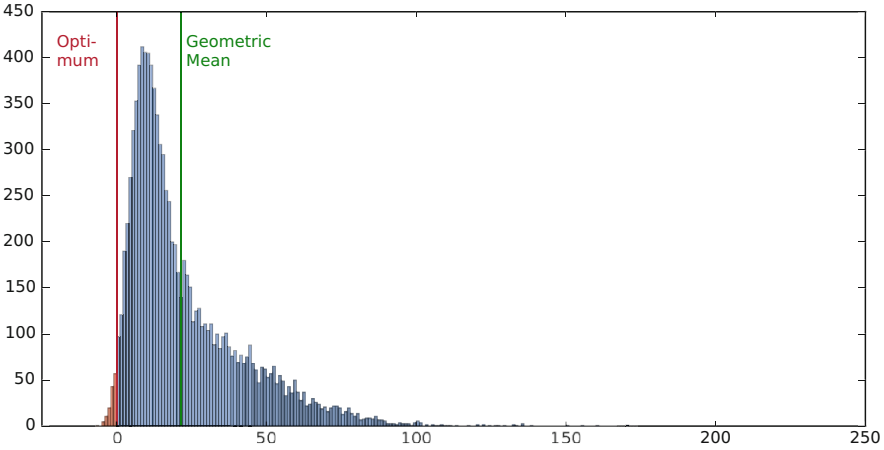

Fig. 7 Histogram of over- and underestimations reported by `aiT` based on 10,000 automatically generated benchmark programs. A value of 0% indicates the optimum (i.e., no analysis pessimism). All (red) values smaller than 0% are underestimations and, consequently, show erroneous analysis reports

no underestimation could be found with the help of *GenE*. With regard to the fact that developers use static WCET analyses for highly safety-critical systems, the benchmark generator *GenE* is a suitable software tool for testing analyzers and increasing their quality.

3.7 Related Work and Generators in the GenE Family

The original idea for the development of the *GenE* benchmark generator is based on the `Csmith` tool [34]: `Csmith` generates programs in the C programming language (i.e., C99) for the purpose of stress-testing compilers. Using the benchmarks generated by `Csmith`, 325 previously unknown bugs were revealed in both open-source and commercial C compilers. These results emphasize the relevance of such structured testing tools. With the same intention and the focus on evaluating bug-detection tools, the program `Bug-Injector` [11] produces benchmarks by relying on bug templates. In contrast to these tools, *GenE* targets comprehensive evaluations of static WCET analyzers.

Besides the original *GenE* tool for the main aim of WCET analysis, we developed *GenEE*, a benchmark generator that specifically targets WCEC analyses [7]. Furthermore, we proposed *Taskers*, a generator for whole real-time systems with multiple tasks [6].

3.7.1 Making Use of Analysis Pessimism on System Level

GenE supports with its specific pattern suites the assessment of individual strengths and weaknesses of WCET analyzers. Although this assessment shows the specific optimization potential of analysis techniques, pessimism remains in the safe upper bounds. Analysis and runtime pessimism (due to not always executing the worst case) then lead to slack resources during the system's runtime (i.e., unused execution time or energy resources). Slack is an undesired property of resource-constrained systems since these systems aim to exploit best the available resources. In order to mitigate this problem of slack and support efficient operation, the dissertation presents the *EnOS* operating-system kernel [29, 30]. The basic idea of *EnOS* is the awareness of mentioned analysis pessimism. That way, *EnOS* supports optimistic scheduling for uncritical tasks (with the use of monitoring techniques) and guarantees the reliable execution under both timing and energy constraints of critical tasks.

4 Conclusion

The dissertation [33] targets the reliable operation of safety-critical systems with both timing and energy constraints. The first main contribution is the program analyzer *SysWCEC* that determines upper energy-consumption bounds of tasks. *SysWCEC* implements an analysis technique for the modeling of temporarily active power consumers. Furthermore, the analyzer accounts for the scheduling semantics with fixed priorities, the tasks' use of operating-system resources (e.g., mutexes), synchronous task activations, and asynchronous interrupts.

Determining the degree of analysis pessimism is a fundamental problem in the domain of worst-case analyses. The dissertation solves the problem of evaluating and validating WCET analysis tools by means of the *GenE* benchmark generator. *GenE* combines small program patterns while keeping track of program-flow facts and, thereby, generates new complex benchmarks, whose worst-case behavior is known. The fact that *GenE*'s generated benchmarks helped to discover previously undetected software bugs in a commercial WCET analysis tool emphasizes the relevance of such structured analysis-testing tools.

The source-code repositories of *SysWCEC*, *GenE*, *Metrics*^{WCA}, and *EnOS* are available online:
https://gitlab.cs.fau.de/syswcec
https://gitlab.cs.fau.de/gene
https://gitlab.cs.fau.de/enos

(continued)

Dataset that helped to reveal software bugs in the `aiT` tool:
https://www4.cs.fau.de/Research/GenE/

References

1. AbsInt: aiT WCET analyzers. https://www.absint.com/ait/
2. Bachmann, O., Wang, P.S., Zima, E.V.: Chains of recurrences—a method to expedite the evaluation of closed-form functions. In: Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC '94), pp. 1–8 (1994)
3. Chu, D.H., Jaffar, J.: Symbolic simulation on complicated loops for WCET path analysis. In: Proceedings of the 9th International Conference on Embedded Software (EMSOFT '11), pp. 319–328 (2011)
4. Cohen, A. et al.: Inter-disciplinary research challenges in computer systems for the 2020s. Tech. rep., USA (2018)
5. Dietrich, C., Wägemann, P., Ulbrich, P., Lohmann, D.: SysWCET: Whole-system response-time analysis for fixed-priority real-time systems. In: Proceedings of the 23nd Real-Time and Embedded Technology and Applications Symposium (RTAS '17), pp. 37–48 (2017)
6. Eichler, C., Distler, T., Ulbrich, P., Wägemann, P., Schröder-Preikschat, W.: TASKers: A whole-system generator for benchmarking real-time-system analyses. In: Proceedings of the 18th International Workshop on Worst-Case Execution Time Analysis (WCET '18), pp. 6:1–6:12 (2018)
7. Eichler, C., Wägemann, P., Schröder-Preikschat, W.: GenEE: a benchmark generator for static analysis tools of energy-constrained cyber-physical systems. In: Proceedings of the 2nd Workshop on Benchmarking Cyber-Physical Systems and Internet of Things (CPS-IoTBench '19) (2019)
8. Falk, H., Altmeyer, S., Hellinckx, P., Lisper, B., Puffitsch, W., Rochange, C., Schoeberl, M., Sørensen, R., Wägemann, P., Wegener, S.: TACLeBench: a benchmark collection to support worst-case execution time research. In: Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis (WCET '16), pp. 1–10 (2016)
9. Gustafsson, J., Betts, A., Ermedahl, A., Lisper, B.: The Mälardalen WCET benchmarks: Past, present and future. In: Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET '10), pp. 137–147 (2010)
10. Huber, B., Prokesch, D., Puschner, P.: 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 (LCTES '13), pp. 163–172 (2013)
11. Kashyap, V., Ruchti, J., Kot, L., Turetsky, E., Swords, R., Pan, S.A., Henry, J., Melski, D., Schulte, E.: Automated customized bug-benchmark generation. In: Proceedings of the 19th International Working Conference on Source Code Analysis and Manipulation (SCAM '19), pp. 103–114 (2019)
12. Knoop, J., Kovács, L., Zwirchmayr, J.: WCET squeezing: On-demand feasibility refinement for proven precise WCET-bounds. In: Proceedings of the 21st Conference on Real-Time Networks and Systems (RTNS '13), pp. 161–170 (2013)
13. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO '04), pp. 75–86 (2004)
14. Li, Y.T.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. In: ACM SIGPLAN Notices, vol. 30, pp. 88–98 (1995)
15. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **4**, 308–320 (1976)

16. Ouyang, H., Liu, Z., Li, N., Shi, B., Zou, Y., Xie, F., Ma, Y., Li, Z., Li, H., Zheng, Q., Qu, X., Fan, Y., Wang, Z.L., Zhang, H., Li, Z.: Symbiotic cardiac pacemaker. Nat. Commun. **10**, 1821 (2019)
17. Pallister, J., Kerrison, S., Morse, J., Eder, K.: Data dependent energy modeling for worst case energy consumption analysis. In: Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems (SCOPES '17), pp. 51–59 (2017)
18. Puschner, P., Prokesch, D., Huber, B., Knoop, J., Hepp, S., Gebhard, G.: The T-CREST approach of compiler and WCET-analysis integration. In: Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS '13), pp. 33–40 (2013)
19. Raffeck, P., Eichler, C., Wägemann, P., Schröder-Preikschat, W.: Worst-case energy-consumption analysis by microarchitecture-aware timing analysis for device-driven cyber-physical systems. In: Proceedings of the 19th International Workshop on Worst-Case Execution Time Analysis (WCET '19), pp. 6:1–6:12 (2019)
20. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Trans. Am. Math. Soc. **74**(2), 358–366 (1953)
21. Scheler, F.: Atomic Basic Blocks: Eine Abstraktion für die gezielte Manipulation der Echtzeitsystemarchitektur. Ph.D. Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, Technische Fakultät (2011)
22. Scheler, F., Schröder-Preikschat, W.: The real-time systems compiler: migrating event-triggered systems to time-triggered systems. Softw. Practice Exp. **41**(12), 1491–1515 (2011)
23. Sieh, V., Burlacu, R., Hönig, T., Janker, H., Raffeck, P., Wägemann, P., Schröder-Preikschat, W.: An end-to-end toolchain: from automated cost modeling to static WCET and WCEC analysis. In: Proceedings of the 20th International Symposium on Real-Time Distributed Computing (ISORC '17), pp. 1–10 (2017)
24. Wägemann, P., Dietrich, C., Distler, T., Ulbrich, P., Schröder-Preikschat, W.: Whole-system WCEC analysis for energy-constrained real-time systems (artifact). Dagstuhl Artifacts Series **4**(2), 7:1–7:4 (2018)
25. Wägemann, P., Dietrich, C., Distler, T., Ulbrich, P., Schröder-Preikschat, W.: Whole-system worst-case energy-consumption analysis for energy-constrained real-time systems. In: Proceedings of the 30th Euromicro Conference on Real-Time Systems (ECRTS '18), vol. 106, pp. 24:1–24:25. Dagstuhl (2018)
26. Wägemann, P., Distler, T., Eichler, C., Schröder-Preikschat, W.: Benchmark generation for timing analysis. In: Proceedings of the 23rd Real-Time and Embedded Technology and Applications Symposium (RTAS '17), pp. 319–330 (2017)
27. Wägemann, P., Distler, T., Hönig, T., Janker, H., Kapitza, R., Schröder-Preikschat, W.: Worst-case energy consumption analysis for energy-constrained embedded systems. In: Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS '15), pp. 105–114. IEEE, Piscataway (2015)
28. Wägemann, P., Distler, T., Hönig, T., Sieh, V., Schröder-Preikschat, W.: GenE: A benchmark generator for WCET analysis. In: Proceedings of the 15th International Workshop on Worst-Case Execution Time Analysis (WCET '15), vol. 47, pp. 33–43 (2015)
29. Wägemann, P., Distler, T., Janker, H., Raffeck, P., Sieh, V.: A kernel for energy-neutral real-time systems with mixed criticalities. In: Proceedings of the 22nd Real-Time and Embedded Technology and Applications Symposium (RTAS '16), pp. 25–36 (2016)
30. Wägemann, P., Distler, T., Janker, H., Raffeck, P., Sieh, V., Schröder-Preikschat, W.: Operating energy-neutral real-time systems. ACM Trans. Embedded Comput. Syst. **17**(1), 11:1–11:25 (2018)
31. Wägemann, P., Distler, T., Raffeck, P., Schröder-Preikschat, W.: Towards code metrics for benchmarking timing analysis. In: Proceedings of the 37th Real-Time Systems Symposium Work-in-Progress Session (RTSS WiP '16) (2016)
32. Wilhelm, R. et al.: The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans. Embedded Comput. Syst. **7**(3), 1–53 (2008)

33. Wägemann, P.: Energy-constrained real-time systems and their worst-case analyses. Ph.D. Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) (2020). https://nbn-resolving.org/urn:nbn:de:bvb:29-opus4-146935
34. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Proceedings of the 32nd Conference on Programming Language Design and Implementation (PLDI '11), pp. 283–294 (2011)