TASKers: A Whole-System Generator for Benchmarking Real-Time–System Analyses

Christian Eichler
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Tobias Distler
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Peter Ulbrich
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Peter Wägemann
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Wolfgang Schröder-Preikschat
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Abstract

Implementation-based benchmarking of timing and schedulability analyses requires system code that can be executed on real hardware and has defined properties, for example, known worst-case execution times (WCETs) of tasks. Traditional approaches for creating benchmarks with such characteristics often result in implementations that do not resemble real-world systems, either due to work only being simulated by means of busy waiting, or because tasks have no control-flow dependencies between each other. In this paper, we address this problem with TASKers, a generator that constructs realistic benchmark systems with predefined properties. To achieve this, TASKers composes patterns of real-world programs to generate tasks that produce known outputs and exhibit preconfigured WCETs when being executed with certain inputs. Using this knowledge during the generation process, TASKers is able to specifically introduce inter-task control-flow dependencies by mapping the output of one task to the input of another.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases benchmarking real-time–system analyses, task-set generation, whole-system generation, static timing analysis, WCET analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2018.6

1 Introduction

Timing and schedulability analyses are essential steps in the development process of a real-time system, allowing developers to ensure, for example, that all tasks meet their deadlines. As analysis runtimes increase with system complexity [6], such analyses usually cannot explore the entire problem space [15], but in part need to make conservative assumptions on system behavior to be feasible and to complete within an acceptable amount of time. When determining upper bounds for the response times of tasks in a system, a typical approach, for example, is to use pessimistic estimates of operating system overheads, because precisely determining their influence often is too expensive [3, 18]. For the same reason, when analyzing the schedulability of a task set, many analysis algorithms assume the absence of inter-task dependencies [1, 4, 5]. Relying on these kinds of assumptions, on the one hand, ensures feasibility, but on the other hand typically also prevents timing and schedulability analyses from producing exact results. This pessimism entails two main consequences: First, with different analysis techniques possibly providing different results, it creates the need to assess the accuracy of each method, both on an absolute scale as well as in relation to alternative techniques. Second, the fact that analysis results are partially based on assumptions makes it necessary to verify whether the guarantees determined still hold when the system is executed on the targeted platform. Both problems can be addressed by benchmarks that provide known baselines, for example, by conducting evaluations in which system code runs on real hardware and execution times are first determined by measurement and then compared to the predicted results.

Comprehensive evaluations require a large number of benchmarks with varying properties and therefore, in general, cannot be performed using real-world programs with given temporal properties only. As a result, it is essential to rely on generated benchmark systems which, to obtain meaningful results, must be configurable yet resemble actual systems as closely as possible. Consequently, such generative benchmarking approaches should, in the optimal case, provide traceable and reproducible inter-task control-flow dependencies, include scheduling and preemption effects, and consider interference caused by the hardware (e.g., caches).

In this paper, we focus on an essential step towards this ultimate goal: The automatic generation of realistic benchmark-system implementations with predefined properties. Most notably, the generated systems have to meet the following requirements: (1) The task implementations of these systems should resemble real-world task programs; in particular, they must not only simulate work by performing busy waiting [10, 14, 23]. (2) The worst-case execution times (WCETs) and the number of generated tasks need to be configurable to be able to create customized benchmark systems that are tailored to specific use cases. Their execution time must vary for different inputs [24, 25], as it is the case for most real-world tasks. (3) The tasks of a generated system need to have dependencies between each other, for example, due to one task's output serving as input for a subsequent task. Inter-task dependencies usually pose a challenge to static analysis techniques and therefore are an ideal means to evaluate the individual strengths and weaknesses of different methods, especially in the context of whole-system timing analysis [6].

To provide the properties discussed above, we designed and implemented TASKers, an approach to automatically generate realistic system implementations that serve as a reliable baseline for benchmarking static analyses. Given a predefined platform and timing configuration, TASKers combines different code patterns that are typical for real-world programs to create a task-set implementation whose tasks match the specified WCETs when being executed with known (configurable) worst-case input values. Starting the system with other input values, on the other hand, results in different (lower) execution times due to task implementations comprising multiple possible program paths. During the task-set generation process, TASKers connects tasks by precedence constraints and thereby introduces control-flow dependencies. Using the generated task set, in a final step TASKers then creates the entire executable benchmark system, including an OSEK-compliant operating system [20].

In summary, this paper makes the following contributions: (1) It presents the TASKers approach for generating benchmarks systems with predefined task WCETs, inter-task control-flow dependencies, and a defined overall worst-case response time (WCRT, the time between the release of an event and the response to it). (2) It discusses details of our current prototype, the TASKers tool. (3) It evaluates the TASKers tool on an ARM Cortex-M4 platform.

The remainder of this paper is structured as follows: Section 2 details the challenges arising from generating complex task sets that resemble real-world systems. Section 3 describes our TASKers approach of generating benchmark programs comprising whole real-time systems. Section 4 presents our prototype that is used for the evaluation in Section 5. Section 6 discusses related work. Finally, Section 7 outlines future work and concludes.

2 Problem Statement

In this section, we discuss two core challenges of generating realistic benchmark systems: (1) Sound synthesis of configurable whole-system implementations that are composed of tasks with inter-task control-flow dependencies and (2) generation of code for said systems that conforms to predefined temporal properties. To complete the problem statement, we furthermore provide details on TASKERS' underlying system model and assumptions.

Challenge #1: Configurable Systems Synthesis with Inter-Task Dependencies

The typical approach to model benchmark systems by artificially consuming execution time (i.e., busy waiting) is precise in time and simple to configure [4, 5, 18]. However, due to the absence of executable code, this approach is inapplicable for the assessment of static analysis techniques. Furthermore, it excludes effects that result from task interactions. For example, given a producing and a consuming task, the consumer's local worst-case path may be infeasible combined with the producer's worst-case path due to mutually exclusive effects. Contrary, evaluation schemes based on real-world systems, or parts of them, are universally applicable as benchmarks. In addition, they inherently incorporate inter-task effects and thus resemble the actual system behavior much more accurately. However, in general, it is challenging to manipulate the temporal properties of functional code to a given specification; not to mention tuning the system's global worst-case path. The resulting problem is, therefore, the generation of benchmark systems with configurable temporal properties that feature realistic precedence constraints and dependencies and provide a common baseline for the assessment of both runtime evaluation and static code analysis.

Our approach: We synthesize evaluation systems with given utilization, number of tasks, and worst-case properties under consideration of target platform and real-time operating system used. To incorporate inter-task dependencies, we rely on task-dependency graphs, which are used to form task systems that fulfill the configured temporal parameters. Therefore, we derive the tasks' WCETs as well as an overall worst-case path budget, which is the result of their composition.

Challenge #2: WCET-Adhering Code Generation

A sound composition and code generation for non-trivial benchmark systems with numerous tasks and interdependencies is challenging: First of all, paths that are locally feasible may become globally infeasible, changing the costs of the global worst-case path. Similarly, inter-task dependencies, such as shared memory, may further affect individual execution costs. Consequently, path budgets and task WCETs must be adapted to compensate for these effects without jeopardizing the overall soundness. These effects virtually impede a straight-line code generation for individual tasks.

Our approach: From the generated task set, we derive worst-case paths and synthesize operational code (i.e., resembles functional code) along with the associated worst-case inputs/outputs. We use a linear regression approach to model the relationship among desired path length and the actual WCET (i.e., number of instructions). In the generation process, we assemble the tasks' implementations to be longest if executed with the preselected worst-case inputs while ensuring that all other paths are shorter. To verify the actual WCETs and the resulting WCRTs, we leverage knowledge of the generated system to trigger and measure local and global worst-case paths at runtime on the target platform. Potential deviations are fed back until all specifications are met.

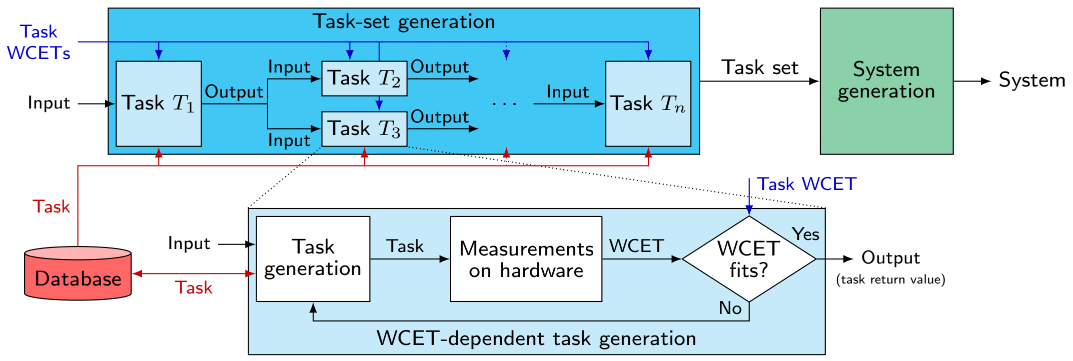

Figure 1 Overview of the TASKers approach

System Model and Assumptions

TASKers targets embedded real-time systems with a single processing core and an operating system that complies with the OSEK standard [20] (i.e., uses fixed-priority scheduling). Consequently, we make the following two assumptions:

(1) The generated task set is OSEK BCC1 compliant. That is, tasks are executed run-to-completion (i.e., non-blocking) in the absence of asynchronous interrupts and events. TASKers generates tasks and task sets under consideration of the target system's scheduling semantics such that their schedule at least becomes deterministic in the worst case (i.e., the worst-case path is triggered). This assumption is not a general limitation: more elaborate operating systems and scheduling variants can be incorporated, as long as the worst-case behavior can be mapped to a defined, deterministic equivalence class of schedules.

(2) Emanating from a given state, the processor is deterministic for any consecutive code sequence. That is, a task's WCET can be measured by the worst-case inputs. Even though this might seem like a severe limitation at first, the assumption of an adequately controllable hardware state is entirely feasible: TASKers exercises control over all code that is executed. This includes, in particular, all initialization processes, whose appropriate size ensures, for example, a known cache state [21]. Sufficiently predictable hardware architectures that feature, for example, an LRU cache replacement strategy are generally available in the embedded domain (e.g., Infineon's Aurix lineup).

3 The TASKers Approach

In this section, we present the whole-system generator TASKers and discuss how it solves the challenges identified in Section 2. In a nutshell, TASKers constructs operational real-time systems by combining generated tasks, thereby ensuring certain properties of both the tasks and the overall system. Figure 1 outlines the three basic steps of the system-generation process: (1) In the first step, TASKers randomly generates a directed acyclic dependency graph based on the number of tasks, their WCETs, and the input value provided by the user. The generated graph describes the producer-consumer relationship and therefore the precedence constraints, while its acyclicity guarantees that there are no cyclic dependencies and thus all dependencies can be fulfilled. These inter-task dependencies mimic the behavior of real-world applications, such as the dependence on a value that is read from a sensor [23]. (2) In the second step, TASKers generates the code for each task in such a way that a task's worst-case path is either triggered by the user-provided input (for the initial task) or by the calculated output of the task's predecessor(s) in the dependency graph. To create a task

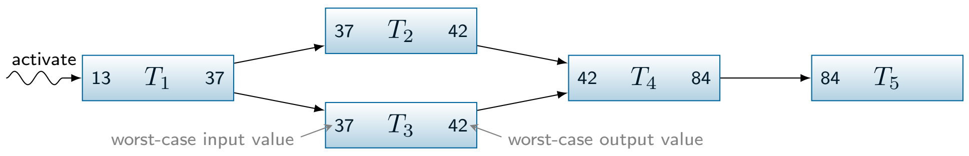

Figure 2 Example of a dependency graph for a task set with tasks T_1, ..., T_5

implementation with a predefined WCET, TASKERS first generates a program whose longest path comprises a specific number of instructions and then determines the actual WCET of the program by measurement on the target hardware. In case the measured WCET does not yet match the intended WCET, in an iterative process, the generator further refines the program by adding or removing instructions. During the generation process, TASKERS stores information about programs and their measured WCETs in a database to speed up the creation of subsequent tasks. (3) In the final step, TASKERS uses the previously generated task set to produce the final, fully operational operating-system binary.

Solution #1: Composing Whole-System Benchmarks with Known Properties
In order to create realistic benchmark systems with inter-task control-flow dependencies, TASKERS starts the generation process by constructing a directed acyclic graph that models all the dependencies between tasks in the final system. As illustrated in Figure 2, for each task this graph also contains information on (1) the input value that should trigger the task's worst-case path and (2) the corresponding output value to be produced for this input. Consequently, TASKERS is able to introduce control-flow dependencies between tasks by mapping the worst-case output value of one task to the worst-case input value of another task. Using the worst-case input value specified by the user, the initial task (i.e., T_1 in the example given in Figure 2) of a generated set is activated through an external signal.

Having constructed the dependency graph, TASKERS then starts to generate code for the individual tasks that meet the requirements specified in the graph. For this step, TASKERS relies on a modified version of the task generator GENE [24], a tool that automatically composes program patterns of real-world tasks to create realistic programs with input-dependent execution times. Most importantly, the programs generated by GENE have known worst-case paths of configurable length (i.e., the number of executed instructions) that are triggered by user-specified worst-case input values. Due to GENE's task generation process being unable to provide specific worst-case output values, we extend the generation approach for TASKERS by introducing a mechanism to track the range of values every variable may have over the course of a task's execution. Knowing the contents of variables for the worst-case input value, TASKERS therefore is able to force a task program to produce a predefined, nevertheless input-dependent, worst-case output value by making the program return the value of an input-dependent variable plus/minus a suitable constant.

Solution #2: Generating Task Implementations with Predefined WCETs
As described above, TASKERS' task generator produces task implementations with a configurable number of instructions on the worst-case path. However, with different instructions usually having different execution times, additional measures are necessary to obtain programs that exhibit predefined WCETs (in terms of milliseconds). For this purpose, TASKERS uses an iterative process that repeatedly adjusts the *path budget* for a task (i.e., the length of the worst-case path in terms of instructions) until the actual WCET of the generated implementation on the target hardware matches the user-specified WCET for the task.

Having generated a task program based on an initial, estimated path budget, TASKERS executes the program on the target hardware with the predefined worst-case input value and measures the resulting execution time. If the measured WCET deviates from the user-specified WCET by less than a configurable margin (e.g., 0.1%), the task-generation process is complete. Otherwise, TASKers starts an additional iteration and generates another program, this time with a different path budget. To determine the new path budget, the tool exploits the fact that, for its generated programs, there is a linear correlation between path budget and WCET (see Section 5.1). In particular, TASKers calculates the linear regression between path budget and measured WCET using several interpolation points, and estimates a fitting anchor value for the new path budget. If necessary, the tool further refines the estimated budget in subsequent iterations by repeated sampling close to the current anchor value and enhancing the linear regression.

To speed up the creation of additional tasks, TASKers stores knowledge on the generated task programs and their respective WCETs in a database. As a consequence, if, for example, a subsequent task is requested to have a similar WCET, the tool does not need to start the generation process from scratch, but can continue based on a previously generated program.

4 Implementation

TASKers' task generator is implemented on top of the LLVM compiler infrastructure [17]. The tasks are generated using LLVM's intermediate representation (LLVM IR), a representation closely related to machine code, while still being independent of the concrete target architectures. For maintaining all information during the lowering from LLVM IR to machine code, the tool relies on control-flow relation graphs [12].

The TASKers task generator is a modified and extended version of the GenE tool [24] and constructs tasks by weaving together different code patterns in a systematic fashion that allows the generator to define the worst-case path. To produce realistic task programs, the code patterns used for this purpose closely resemble the typical building blocks of real-world applications. Amongst other things, this includes input-dependent computations, assignments, function calls, conditional branch statements, as well as different shapes of loops. The fact that most code patterns are parameterizable enables TASKers to create complex programs in which, for example, the execution of branches depends on the program's input and the bodies of loops may contain other, nested code patterns, possibly even other loops. Prior to the start of the generation process, a TASKers user is able to influence the composition of the resulting benchmark by configuring the probability with which each code pattern is selected and woven into the program. This way, a user can rely on TASKers to generate benchmark-system implementations that possess the characteristics of real-time systems from a particular domain (e.g., digital-signal-processing applications).

In contrast to GenE, TASKers' task generator does not use the path budget as seed for pseudorandomly selecting the next program pattern to weave into the task program. Instead, TASKers limits the impact small changes to the path budget have on the structure of the generated program, thereby ensuring that two programs resemble each other if they were generated with similar (but different) path budgets. As a key benefit, this approach improves the linearity between the path budget and a task implementation's WCET, and therefore significantly speeds up the process of creating a program with the requested WCET.

The task generator is integrated into and driven by a Python framework that, in addition, takes care of producing the task set, building the system binary, measuring on the target hardware and postprocessing of the measured values. The final system binary is assembled on basis of *d*OSEK [11], an OSEK-compliant real-time operating system.

5 Evaluation

In this section, we move on to the evaluation of TASKers. First, we assess our approach to correlate desired path budget and actual WCET of the operational code by linear regression. Based on this, we demonstrate that in the worst case the generated tasks meet specifications and otherwise exhibit realistic behavior. Finally, we showcase the performance and worst-case compliance of a whole system and its suitability as a benchmarking baseline.

Experimental Setup
We used an Infineon XMC4500 development board with an ARM Cortex-M4 processor as an available hardware platform with the instruction timing, pipeline behavior, and memory-access latencies being sufficiently predictable and well documented [13]. The 32-bit processor runs at 120 MHz and features a 3-stage pipeline, 1024 KB flash memory with 4 KB instruction cache (2-way set associative, LRU cache-replacement policy). Time measurements were conducted using the cycle-accurate counters (i.e., DWT) provided by the Cortex-M4 core. For the measurements, we applied our system model as described in Section 2. All tasks are generated using the default configuration of TASKers' task generator (i.e., all code patterns are enabled and use their default probabilities).

5.1 Correlation of Path Budget and WCET

TASKers' task generator exploits the linear correlation between the path budget and the WCET of the resulting operational code to estimate the budget required to generate a fitting task. To confirm the assumption of linear correlation between path budget and a task's WCET, for our first experiment we generated 20,000 tasks with different path budgets, while retaining the generator's other parameters, and determined the tasks' WCETs. Figure 3 illustrates the correlation between the path budget and the tasks' WCET: Each data point (dot) represents a single task whose WCET is plotted against its predefined budget. The regression used by TASKers to estimate a value for the path budget is given as a line. The adjacency and concentration of the measured data points close to the regression line substantiate the assumption of the linear correlation between path budget and WCET and thus TASKers' approach to generating tasks with predefined WCETs.

5.2 Temporal Behavior and Realism of Tasks

In a next step, we evaluated the accuracy of the operational code generation itself. Therefore, we generated tasks with budgets in the range from 1 ms to 5 ms and a maximum deviation of 0.1% and measured the actual WCET by triggering the generated worst-case path at

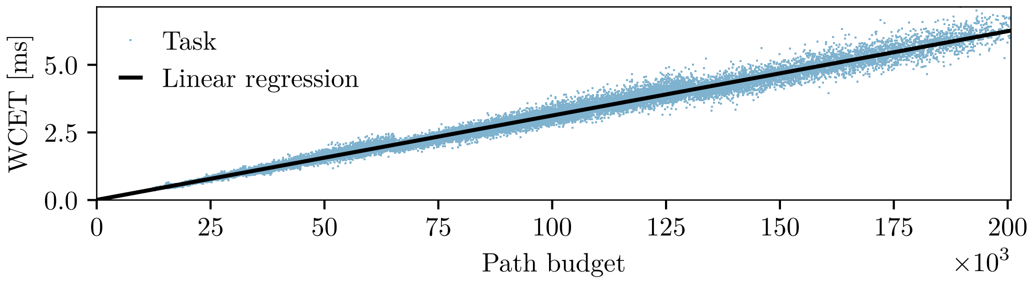

Figure 3 Linear correlation between path budget and the resulting task's WCET

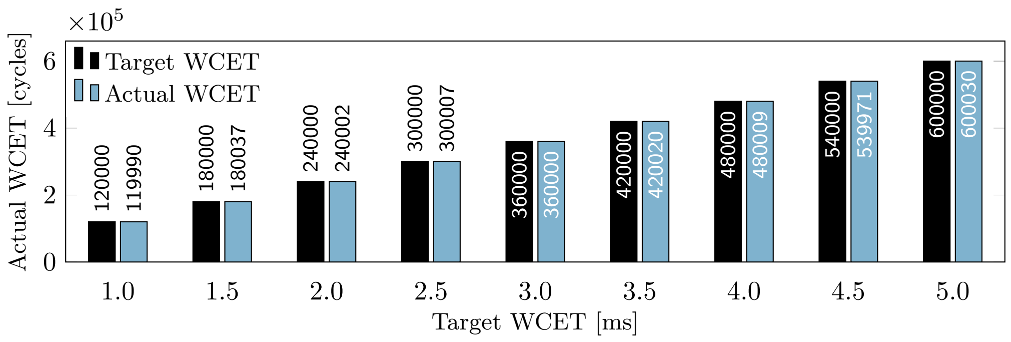

Figure 4 Comparison between generated and target WCETs

runtime. Figure 4 shows the results of this evaluation: For all 9 tasks, the actual WCET fits the predefined budget with great accuracy. The maximum deviation in this experiment was 0.02% or 37 cycles (1.5 ms task), well below the configured threshold of 0.1%. For all practical purposes, TASKERS' code generation accuracy per task is more than sufficient for the composition of whole-system benchmarks.

The tasks generated by TASKERS show a WCET that is defined at generation time and can be triggered by a user-defined input. Moreover, as identified in Section 2, tasks should in practice also show input-dependent variations of their execution times, adding a jitter component to the benchmark. To assess this aspect of the code generation, we measure the 3 ms-task's execution time for 2^{24} = 16,777,216 different input values; Figure 5 graphs the resulting distribution. First of all, the traversal of the worst-case execution path is triggered by 19 different input values, including the designated worst-case input value used during the generation process. The WCET (i.e., the execution time of the task's worst-case execution path) is 360,000 cycles (3.0 ms at 120 MHz) and matches the predefined WCET of 3.0 ms. Moreover, the task exhibits a wide range (0.2 to 3 ms) and distribution of execution times. The variation also indicates the multitude of different paths through the task.

In summary, the generated operational code complies with the preassigned temporal properties with high accuracy and, at the same time, resembles real-world execution behavior.

5.3 Whole-System Temporal Behavior and Worst-Case Path

This final evaluation is devoted to the temporal behavior of a whole system. Therefore, we took as an example a task set consisting of five interdependent tasks and leveraged the knowledge about the generated system to incorporate the overheads induced by the underlying operating system. To stay unbiased, we randomly chose the tasks' WCETs

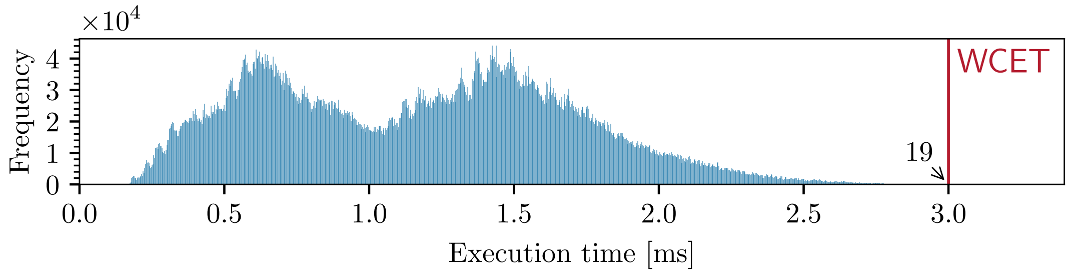

Figure 5 Variation of the execution time for a single task with varying input values

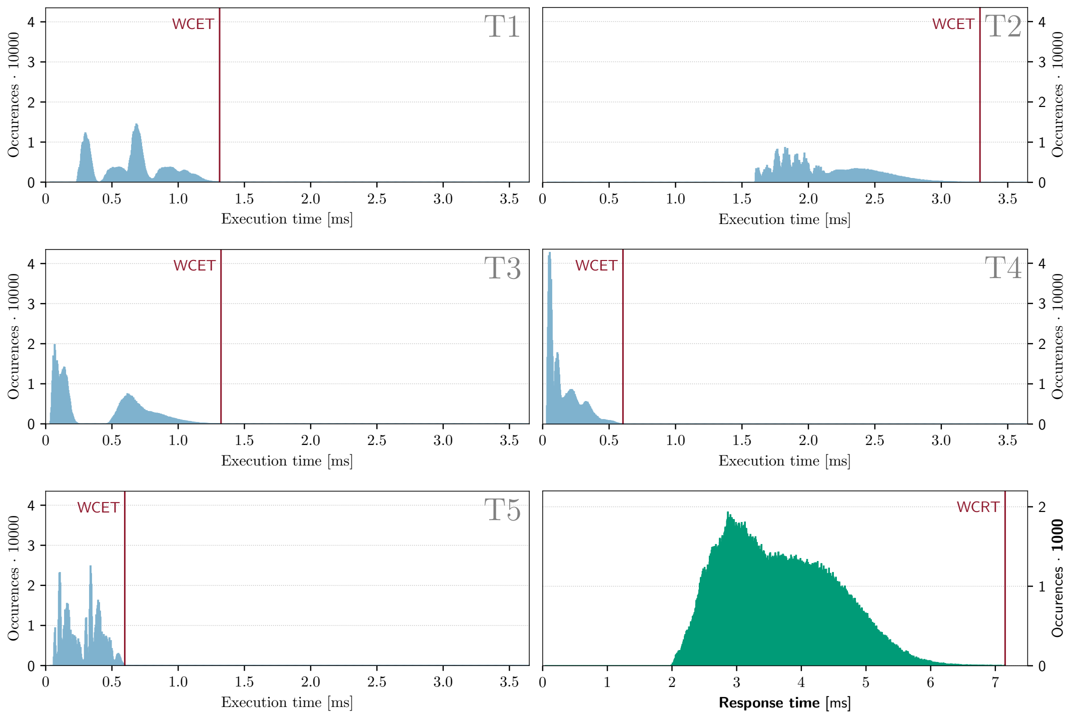

Figure 6 Execution and response times for task set of five tasks $T_1, ..., T_5$

(71,637 cycles, 72,519 cycles, 158,795 cycles, 394,958 cycles, 157,532 cycles for tasks $T_1, ..., T_5$) using the UUniFast [2] algorithm. For the generated benchmark system, we measured the individual execution times of all five tasks, as well as the overall response times for different input values. Figure 6 gives the frequencies of the measured execution and response times for $2^{20} = 1{,}048{,}576$ randomly selected input values. All tasks $T_1, ..., T_5$ exhibit varying execution times, ranging from 4,257 cycles ($\sim$ 35.5 µs, T_2) to 394,958 cycles ($\sim$ 3.3 ms, WCET of T_3), again substantiating the variety of local execution paths. Except for T_1, each task's execution path depends on the output of the previous task's calculation, the high distribution of execution times indicates a substantial variation of global control-flow paths through the generated system that depends on the respective input and output values.

The longest execution path through the task set (calculated as the sum of the corresponding tasks' execution times) takes 7,129 µs and is traversed whenever the predefined worst-case input value is passed to T_1. Incorporating the overheads induced by the operating system, the system's actual response time is higher. In case of this experiment, the actual worst-case response time measured by TASKers is 7,151 µs; 22 µs higher than the lower bound (i.e., the summation of individual execution times).

In summary, our evaluation demonstrates that the overall system also conforms to predefined temporal properties and exhibits known WCRTs.

6 Related Work

To our knowledge, TASKers is the first whole-system generator for benchmarking real-time systems. The generator is able to create tasks with arbitrarily configurable WCETs and complex inter-task dependencies. The closest related work to ours is the task-set–generation approach from de Bock et al. [4]. They use existing benchmarks from the TACLeBench suite [8], which are written in C, and execute several programs in a loop in order to achieve a predefined execution time. In contrast to their approach, we do not rely on already available benchmarks but generate each task from scratch using a low-level representation (i.e., LLVM IR) and consequently do not depend on the immutable execution times of existing programs. Generating both the tasks as well as the enclosing system from scratch has the major benefit of enabling TASKers to insert inter-task dependencies on a fine-grained level. Additionally, TASKers lays the foundation for automatically creating programs to benchmark memory-consumption and caching-behavior analysis, which cannot be achieved when relying on memory consumption and access patterns from existing benchmark programs written in a high-level programming language. In contrast, using LLVM's intermediate representation results in our benchmarks being more resilient against compiler optimizations compared to the TACLeBench suite [24, 25]. The most important difference is that, even though TASKers' generated tasks have varying execution times, the worst-case path of the overall system is known. Executing this non-trivial path with the predefined worst-case input leads to the WCRT of the task set, thereby providing a baseline for timing-analysis approaches.

Several real-world benchmarks and suites are used for benchmarking real-time system analyses [8, 9, 10, 14, 16, 19]. However, in order to evaluate approaches on a global scale, the ground truth, such as the worst-case path, is inevitable, which can never be safely determined when trying to extract it from existing code [22]. Instead, we solve the problem of baselines by *generating* the entire executable real-time system with known properties.

Tools exist that output the parameters of tasks, such as UUniFast [2] or the task-set generator from Emberson et al. [7]. These generators are used for the theoretic evaluation of scheduling algorithms since they do not produce any executable code. TASKers fills this gap by considering these task parameters and producing an executable system for comprehensive evaluations of timing-analysis approaches on real embedded platforms.

7 Conclusion & Future Work

In this paper, we presented the TASKers generator that produces real-time systems with known properties for the purpose of benchmarking timing-analysis approaches. TASKers generates entire systems that execute precedence-constrained tasks with configurable WCETs, which are scheduled by their fixed priorities. Since all relevant knowledge about the generated systems is either directly available or can be determined by measurement (e.g., a system's WCRT), TASKers enables a comprehensive evaluation of timing and scheduling-analysis techniques on an absolute scale. Our evaluation confirms TASKers' accuracy, showing that the actual WCETs of generated tasks differ less than 0.1% from the WCETs requested.

As part of future work, we will extend the current prototype of TASKers to make it more aware of the target platform's micro-architectural properties, and thereby further challenge analysis approaches. Specifically, we will integrate challenging code patterns to generate benchmarks that target whole-system cache and pipeline analyses.

The source code of TASKers *is available at*: `https://gitlab.cs.fau.de/taskers`

Acknowledgements This work is supported by the German Research Foundation (DFG), in part by Research Grants no. SCHR 603/9-2 (AORTA), SCHR 603/13-1 (PAX), the 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.

References

1 M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo. Energy-aware scheduling for real-time systems: A survey. *ACM TECS*, 15(1), 2016.

2 E. Bini and G. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30, 2005.

3 B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser. Timing analysis of a protected operating system kernel. In *Proc. of RTSS '11*, 2011.

4 Y. De Bock, S. Altmeyer, J. Broeckhove, and P. Hellinckx. Task-set generator for schedulability analysis using the taclebench benchmark suite. In *Proc. of EWiLi '16*, 2016.

5 A. Burns and R. Davis. Mixed criticality systems - a review. Technical report, Department of Computer Science, University of York, 2018.

6 C. Dietrich, P. Wägemann, P. Ulbrich, and D. Lohmann. SysWCET: Whole-system response-time analysis for fixed-priority real-time systems. In *Proc. of RTAS '17*, 2017.

7 P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *Proc. of WATERS '10*, 2010.

8 H. Falk et al. TACLeBench: A benchmark collection to support worst-case execution time research. In *Proc. of WCET '16*, 2016.

9 J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks: Past, present and future. In *Proc. of WCET '10*, 2010.

10 M. Guthaus et al. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of WWC '01*, 2001.

11 M. Hoffmann, F. Lukas, C. Dietrich, and D. Lohmann. dOSEK: The design and implementation of a dependability-oriented static embedded kernel. In *Proc. of RTAS '15*, 2015.

12 B. Huber, D. Prokesch, and P. Puschner. Combined WCET analysis of bitcode and machine code using control-flow relation graphs. In *Proc. of LCTES '13*, 2013.

13 Infineon Technologies AG. XMC4500 reference manual, 2012.

14 F. Kluge, C. Rochange, and T. Ungerer. Emsbench: Benchmark and testbed for reactive real-time systems. *Leibniz Trans. on Embedded Systems*, 4(2), 2017.

15 J. Knoop, L. Kovács, and J. Zwirchmayr. WCET squeezing: On-demand feasibility refinement for proven precise WCET-bounds. In *Proc. of RTNS '13*, 2013.

16 S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmarks for free. In *Proc. of WATERS '15*, 2015.

17 C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of CGO '04*, 2004.

18 M. Lv, N. Guan, Y. Zhang, Q. Deng, G. Yu, and J. Zhang. A survey of WCET analysis of real-time operating systems. In *Proc. of ICESS '09*, 2009.

19 F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. De Michiel. PapaBench: A free real-time benchmark. In *Proc. of WCET '06*, 2006.

20 OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, February 2005.

21 Jan Reineke. *Caches in WCET Analysis: Predictability, Competitiveness, Sensitivity.* PhD thesis, Saarland University, 2008.

22 H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. of the AMS*, 1953.

23 P. Ulbrich, R. Kapitza, C. Harkort, R. Schmid, and W. Schröder-Preikschat. I4Copter: An adaptable and modular quadrotor platform. In *Proc. of SAC '11*, 2011.

24 P. Wägemann, T. Distler, C. Eichler, and W. Schröder-Preikschat. Benchmark generation for timing analysis. In *Proc. of RTAS '17*, 2017.

25 P. Wägemann, T. Distler, P. Raffeck, and W. Schröder-Preikschat. Towards code metrics for benchmarking timing analysis. In *Proc. of RTSS WiP '16*, 2016.