Task-Set Generator for Schedulability Analysis using the TACLeBench benchmark suite

Yorick De Bock
imec, IDLab, Faculty of Applied Engineering
University of Antwerp, Belgium
yorick.debock@uantwerpen.be

Sebastian Altmeyer
Faculty of Science
University of Amsterdam, The Netherlands
altmeyer@uva.nl

Thomas Huybrechts
imec, IDLab, Faculty of Applied Engineering
University of Antwerp, Belgium
thomas.huybrechts@uantwerpen.be

Jan Broeckhove
imec, IDLab, Department of Mathematics and Computer Science
University of Antwerp, Belgium
jan.broeckhove@uantwerpen.be

Peter Hellinckx
imec, IDLab, Faculty of Applied Engineering
University of Antwerp, Belgium
peter.hellinckx@uantwerpen.be

ABSTRACT

Currently, real-time embedded systems evolve towards complex systems using new state of the art technologies such as multi-core processors and virtualization techniques. Both technologies require new real-time scheduling algorithms. For uniprocessor scheduling, utilization-based evaluation methodologies are well-established. For multi-core systems and virtualization, evaluating and comparing scheduling techniques using the tasks' parameters is more realistic. Evaluating such scheduling techniques requires relevant and standardised task sets. Scheduling algorithms can be evaluated at three levels: 1) using a mathematical model of the algorithm, 2) simulating the algorithm and 3) implementing the algorithm on the target platform. Generating task sets is straightforward in the case of the first two levels; only the parameters of the tasks are required. Evaluating and comparing scheduling algorithms on the target platform itself, however, requires executable tasks matching the predefined standardised task sets. Generating those executable tasks is not standardized yet.

Therefore, we developed a task-set generator that generates reproducible, standardised task sets that are suitable at all levels. Besides generating the tasks' parameters, it includes a method that generates executables by combining publicly available benchmarks with known execution times. This paper presents and evaluates this task-set generator.

CCS Concepts

•**Computer systems organization** → **Real-time systems;** *Embedded software;* •**General and reference** → *Empirical studies; Measurement;*

Keywords

Real-Time Systems, Embedded Systems, Schedulability Analysis, Benchmarks, Task-Set Generator, Software Tool

EWiLi'16, October 6th, 2016, Pittsburgh, USA.

1. INTRODUCTION

Current evolutions of mechatronics show an exponential growth in the number of embedded systems used for control due to the shift from mechanical control towards electronic control. This evolution creates new opportunities for more advanced software based control. On the downside, it introduces new challenges regarding safety and reliability. Both opportunities and challenges will result in more complex software and hence higher computational requirements. These requirements can be tackled by using state of the art technologies on embedded system. Multi-core processors are a solution to increase the computational power in a single chip. Virtualization techniques can be applied to handle the complexity of the software by decomposing the software into components which can be analysed independently of each other. For real-time systems, most applications are composed of recurring tasks with periods, deadlines and execution times. Such a collection of tasks is called a task set. The schedulability of task sets is very important. Uniprocessor scheduling algorithms are evaluated and compared based on the maximum task set utilization that can be achieved by the scheduler. For multiprocessor scheduling algorithms, however, the complexity of the analysis methodologies increases dramatically. This is due to the concurrent execution of tasks and the indeterminism of most multi-core architectures. Virtualization introduces a two-level hierarchical scheduling structure that renders the traditional analysis inapplicable. The many open issues and the practical importance of these technologies make it a topic of active research.

Parameter-based analysis uses the tasks' parameters during analysis. This results in an one-to-one mapping of task-set and scheduling algorithm; the schedulability can be analysed for a specific task set using a certain scheduling algorithm. The parameter-based analysis is more realistic for multiprocessor scheduling algorithms, and at this moment the only analysis method for the hierarchical scheduling structure in virtualization technology. The schedulability of a task set is the key criterion for the evaluation and comparison of scheduling algorithms. Other criteria such as the number of pre-emptions, energy consumption, scheduler

overhead, cache performance etcetera provide an additional basis for comparison. The schedulability and other criteria can be evaluated at an early stage of the design process of the mechatronic system. At this stage, the complexity and cost are relatively low compared to later stages of the process. However, at the later stages more evaluation criteria can be evaluated and compared with other scheduling algorithms.

The performance of scheduling algorithms can be evaluated on three levels:

- **Formal proof:** schedulability can be formaly proven by the mathematical model of the scheduling technique.
- **Simulation-based analysis:** schedulability is validated based on the simulation model of the scheduling technique. This technique simulates a task set scheduling based on specific input parameters covering the target hardware (number of cores, ...) and the simulator settings (stepsize, simulation time, ...).
- **Implementation-based analysis:** real-time tasks are deployed on the target platform. To schedule the tasks on the target platform, a scheduler and hence a Real-Time Operating System (RTOS) are required. The scheduler uses the scheduling algorithm to define the order of execution of the tasks.

Evaluating scheduling algorithms requires input data. A set of real-time tasks is needed to evaluate and compare different scheduling mechanisms. Depending on the evaluation level the content and format of the tasks and task set differ considerably. At the first two levels, synthetic task sets are required. This implies that the task model should only include the tasks' parameters (Worst-Case Execution Time (WCET), period and deadline) of the different tasks. It is essential that the generated values of those parameters are not biased against any scheduling algorithm. At the third level, however, the task model includes executable code, which has to match the existing task parameters. The executable tasks should be created taking into account the WCET parameter of the task. The WCET of a task is expressed in time units and will differ when deployed on different target platforms. Therefore the task model should be injected by different sets of task code when deployed on different platforms. The latter feature is missing in current evaluations of scheduling algorithms making it hard to compare or reproduce their results across platforms.

Our goal, in this paper, is to create executable tasks, based on the synthetic task set, for a broad set of different architectures and to examine and compare the performance of the scheduling algorithms. The execution time of the tasks will be equal to the WCET of the tasks to test the worst-case scenario for the scheduler.

We present a task set-generator tool, which not only generates synthetic task sets, but also the executable tasks for a given target platform using publicly available benchmark programs. This tool can be used to generate task sets suitable to evaluate and compare scheduling algorithms at all evaluation levels using standardized, reproducible and transparent task-set generation techniques. The task-set generator tool is part of the COBRA framework. A framework to optimize the (worst-case) resource consumption on multiple research levels: timing, scheduling and parallelism.

The remainder of the paper is organized as follows. The COBRA framework is described in Section 2. The related work on generating synthetic task sets and executable tasks is briefly reviewed in Section 3. The task-set generator tool and its different components are discussed in Section 4. Section 5 describes an experimental evaluation of the tool. We conclude the paper and give a brief overview of future research in Section 6.

2. COBRA FRAMEWORK

COBRA (COde Behaviour fRAmework) is an open source framework (Figure 1) that allows developers to optimize (worst-case) resource consumption. Currently it focusses mainly on computational resources, but it will soon be extended with power consumption and other resources to optimize. The COBRA framework combines multiple research domains to optimize the resource consumption on multiple levels (timing, scheduling and parallelism):

- **WCET analysis.** To determine the Worst-Case Execution Time (WCET), different techniques are available. The COBRA framework implements both static and dynamic timing analysis, and even new hybrid techniques which combines the existing static and dynamic analysis techniques. These techniques are essential to characterise the resource consumption.
- **Scheduler optimization** towards real-time performance and/or power consumption. Energy and time (computing units) are both important resources, this part of the framework analyses and optimizes scheduling algorithms towards these resources given a specific application. The analysis methodology to find this optimal scheduling algorithm, includes the three evaluation levels, described in Section 1. This gives us the opportunity to select a set of algorithms at the beginning of the design process and to narrow it down towards the end.
- **Design pattern based performance optimization for multi-core processors.** Executing an application on a multi-core processor can have a speed-up due to parallelism. However finding and implementing the right design patterns to find parallelism is time consuming. This process can be repeated at multiple levels, from code-based parallelism towards instruction-level parallelism. This framework tries to automate this process to find the best design patterns for parallelism at the right level.

As described in Figure 1, the COBRA framework[1] consists of three parts which can be used separately. The first part delivers input for the second part to be manipulated or to be directly used as input data for the experiments in the third part. The second part transforms the input of the first part to specific applications with user-defined parameters to test a specific case for the analysis techniques. The third

[1]The COBRA framework and all of its components, are available at our website (http://cobra.idlab.uantwerpen.be). The task-set generator can be downloaded together with the TACLeBench benchmark programs. This website also provides installation guidelines and a user manual for the task-set generator.

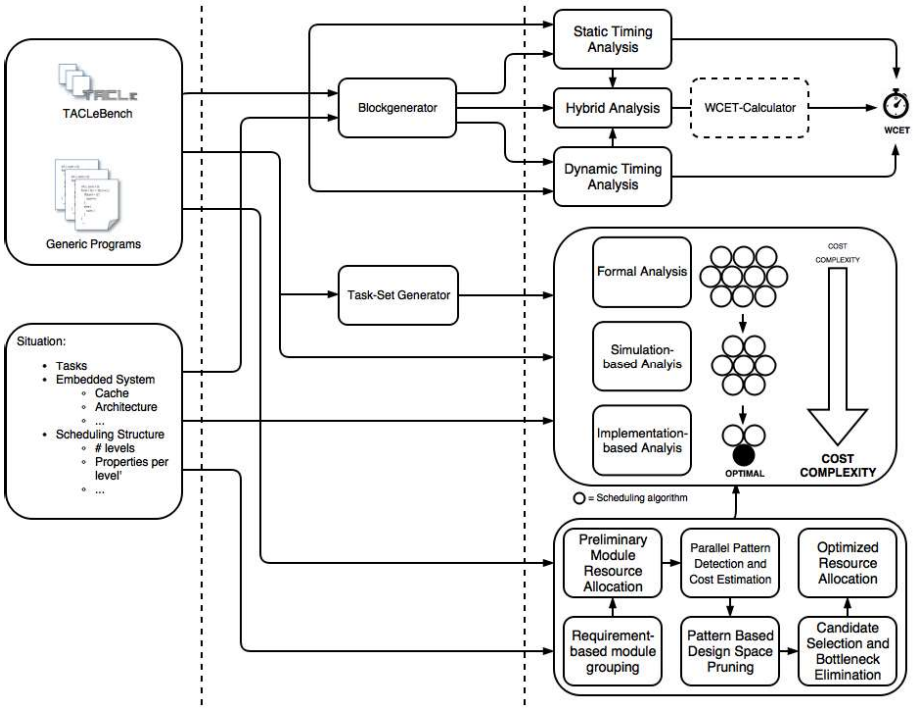

Figure 1: The COBRA Framework: the three parts are visualized by the dotted lines. The first parts, located on the left, contains the different sources of the input data. In the middle, the second parts provides a set of tools which modify the input data of the first parts and provide the output to the analysis tools in the third part. However, the input data of the first part can also be a direct input for the analysis tools in most instances. The third part, on the right, contains the analysis tools. Above the WCET analysis techniques, with the new hybrid technique. In the middle the analysis methodology to find the optimal scheduling algorithm and below the design pattern based performance optimization for multi-core processors.

part analyses and optimizes the resource consumption for the different research domains discussed above.

The input in the first part consists of (1) the TACleBench benchmark suite [4], (2) Generic Input Programs and (3) specific applications given a specific hardware configuration. The second part contains multiple tools to process and modify the input data of the first part. These tools provide their processed output data in the same format for every analysis tool, despite the format of the raw input. Because of this standardization, the only requirement for new analysis tools to be added to the COBRA framework is to support this format. The third part contains for each research domain an analysis tool or combination of tools. Each analysis tool optimizes the resource consumption by implementing new analysis techniques and methodologies.

The *task-set generator* discussed in this paper is part of the COBRA framework and the generated task sets (both synthetic and executable) are used in the multiple research domains explained above (WCET analysis, scheduling optimization and design pattern based optimization for multi-core processors). The task-set generator is a tools that belongs to the second part. It formats, converts and processes raw input data into a standardized format supporting reproducible analysis experiments. Its modus operandi will be described in more detail in this paper.

3. RELATED WORK

Research on task-set generation tools that generate executable tasks is rather limited. TIMES [1], by Amnell et al., is a tool specifically designed for symbolic schedulability analysis and synthesis of executable code with predictable behaviours for real-time systems. It includes a task-set generator to integrate a set of tasks provided by the user into specific hardware architectures. Starting from a set of tasks and their runtimes, it will analyse schedulability given a specific scheduling method and hardware architecture. After successful analysis it will generate wrapping code around predefined tasks to create a compilable source code project for that specific type of hardware.

In [5] Kramer et al. present a new benchmark generator methodology for automotive applications. The automotive industry is characterized by its strong intellectual property (IP) protection. This results in a lack of realistic real-world benchmark applications. Kramer et al. propose to solve this problem by creating new benchmark applications based on code snippets coming from a well protected database containing IP protected automotive applications. The tool however was not yet available at the time of writing.

Wägemann et al. proposed GenE [8], a tool to generate benchmarks for timing analysis. It combines code patterns from real-time applications that are both representative for real-time applications and sufficiently challenging to WCET analysis tools. In addition to the source code, the generator also provides the flow facts of the benchmark. Based

on this information it is straightforward to derive an accurate WCET that can be used as a reference to evaluate and compare the performance of WCET analysis techniques and technologies.

4. TASK-SET GENERATOR TOOL

The tool presented in this paper will generate task sets for the evaluation of scheduling algorithms at the three levels of abstraction in the design process. For formal and simulation based analysis, synthetic task sets are generated given a global utilization range for the task sets. A task set S consists of a set of n independent real-time tasks $\{\tau_1, \tau_2, ..., \tau_n\}$. Let τ_i indicate any given task of the task set. Each task has three parameters: the WCET C, the relative deadline D and the period T. The utilization of a task set is defined as:

$$Ut = \sum_{i=0}^{n} U_i = \sum_{i=0}^{n} \frac{C_i}{T_i} \quad (1)$$

where C_i is the WCET of task τ_i and T_i the period of task τ_i.

For implementation-based analysis, the tasks of the synthetic task sets are extended with an executable instance. These executable tasks are created by combining benchmark programs from the TACLeBench benchmark suite [4]. The benchmark suite is a collection of benchmark programs used to evaluate timing analysis tools. The task-set generator tool calculates for each task a combination of benchmark programs. The summation of the execution time of the selected benchmark programs equals the required execution time of the task (within a pre-defined error margin). The tool distinguishes two types of user-defined input parameters: *task set specific parameters* and *program specific parameters*. The former are used to generate the synthetic task sets, the latter are used to select a set of benchmarks which fits the requirements of the user and the target platform. The selected set is used to calculate the sequence of benchmark programs for each tasks of the generated synthetic task set.

The tool is structured into three major parts which can operate independently. The first part creates the synthetic task sets, the second part selects the combination of benchmark programs for each task and the last part generates the source code and the makefile to create the executable tasks.

4.1 The Synthetic Task-Set Generator

To generate task sets to evaluate and compare scheduling algorithms, the distribution of the utilization of the tasks must be unbiased towards any scheduling algorithm [3]. The generator generates periodic tasks with implicit deadlines. These are tasks with a deadline D equals to the period T which means that the the task releases a job at every time interval T. To generate the synthetic task sets, the tool uses the *task set specific parameters* as input. The following parameters are a minimum set of parameters:

- the range of the task set utilization ($[Ut_{min}, Ut_{max}]$)
- the step value between two utilizations (Ut_{step});
- the number of task sets per utilization (k);
- the number of tasks per task set (n);
- the lower and upper bound on the period (T_l, T_u);
- the level of granularity of the period (T_Δ);
- the seed value for pseudorandom generators. (s)

In a first step, a utilization U_i for each task τ_i in the task set S is defined based on the constraint that $\sum_{i=0}^{n} U_i = U_t$ where U_t is the target utilization. The UUnifast-Discard algorithm is used to generate the task-specific utilizations. The input of this algorithm is the utilization of the task set U_t and the number of tasks in the task set n.

In the second step, the period of each task is defined and the execution time is calculated for each task. The period is randomly selected between the given lower bound T_l and upper bound T_u. Based on the generated period T and task utilization U, the execution time C is calculated. The mininum difference between periods of two different tasks is defined as T_Δ. To evaluate the effect of an input parameter, pseudorandom generators are used to generate these values. This implies that, for example, two identical task sets, in terms of task utilization, can be generated, but with a different lower and upper bound of the period. By keeping all other parameters fixed, all confounding effects are avoided [2].

The number of generated task sets (m) depends on the number of utilization steps and the number of task sets per utilization, $m = \left(\frac{Ut_{max} - Ut_{min}}{Ut_{step}} + 1\right) \times k$. Another advantage of the pseudorandom generator is its ability to reproduce identical tests. An identical set of task sets can be generated by other parties in the same domain when using the same input parameters. The generated synthetic task sets are used to generated the benchmark program sequence in the second part.

4.2 Benchmark Program Sequence

After the synthetic task sets are generated, a benchmark program sequence is calculated for every task. The tool uses the TACLeBench benchmark suite as a source for the benchmark programs [4]. The benchmark programs are formatted using the same code formatting rules, this results in a main function consisting of three function calls. These functions are present in every benchmark program:

- {*benchmark program name*}*_init()*
- {*benchmark program name*}*_main()*
- {*benchmark program name*}*_return()*

The first function initializes the benchmark program, the second executes the main functionality and the third function returns a variable for sanity checks. If the benchmark program is selected for a task, the first and second function execute multiple times in each job of that task. This insures that the benchmark program always has the same input data each time it executes. Furthermore, almost all benchmarks are platform-independent and can be compiled to and evaluated on any kind of target platform. To use the benchmark programs in the task-set generator, information regarding each benchmark must be accessible for the tool. This is realized by a complementary description file for each

benchmark. This description file contains all necessary information of the benchmark (location, execution time,...). Based on the information in the description file and the selection criteria given by the user, the tool checks whether the benchmark program is suitable to be used in the benchmark program sequence. The first round of selection is based on the architecture of the target hardware; if the description file includes timing information (execution cycles) of the benchmark program on the given architecture, the benchmark program is selected. This selection results in a subset of the benchmark programs which are used to calculate the benchmark program sequence for each task. By adding more information about the benchmark programs in their description file, a more precise selection of benchmark programs is possible. To prevent non-reproducible behaviour, each benchmark program should run at least a minimum number of times in a row; due to cache effects and other micro-architectural features, a program's execution time may vary strongly. Yet, when executed in sequence, the execution time eventually stabilizes. Hence, we also derive the minimum number of executions until a stable execution time occurs. This means that when a benchmark program is selected, the number of times a benchmark program is executed lies between its minimum number and infinity. A minimum number of consecutive executions is necessary to get a reproducible execution time. For each benchmark program, the minimum number of executions is different and does not only depends on the code of the benchmark program, but also on the architecture of the target hardware. The value of this minimum number of executions, has to be obtained in a measurement based approach by executing the benchmark program in a loop and varying the loop size. This minimum number of executions is included in the description file of the benchmark program. To calculate the program sequence for each task, we use an Integer Linear Programming (ILP) model [7] to describe the optimization problem. This model tries to match the sum of the benchmark program execution times with the target WCET of the task. The number of times a benchmark program is used, equals or is greater than zero. To build this model, the execution time of the benchmark programs must be known. These are calculated by dividing the number of execution cycles of a benchmark program by the clock frequency of the processor of the target platform. The model must also be aware of the minimum required number of executions of each benchmark program. This is included in the model as an initial cost function. If a benchmark program is selected, a cost c (minimum number of execution multiplied by the execution time of the benchmark program) is added once. The model is represented by the the following equations:

$$\text{maximize} \sum_{i=1}^{k} (c_i y_i + e_i l_i) \tag{2}$$

$$\text{subject to} \sum_{i=1}^{k} (c_i y_i + e_i l_i) \leq E, \qquad i = 1, ..., k \tag{3}$$

$$l_i \in \mathbb{Z}_{\leq 0}, \qquad i = 1, ..., k \tag{4}$$

$$y_i = \begin{cases} 0 \text{ for } l_i = 0 \\ 1 \text{ for } l_i > 0 \end{cases}, \qquad i = 1, ..., k \tag{5}$$

$$y_i \in \{0, 1\}, \qquad i = 1, ..., k \tag{6}$$

where $e_0, ..., e_k$ is the set of execution times of the subset of benchmark programs. The number of times a benchmark program must be executed is represented by $l_0, ..., l_k$, and E represents the targeted execution time of the task. The initial cost of each benchmark program is represented by $c_0, ..., c_k$. The ILP tries to maximize Equation 2, but it is subject to Equations 3-6. Equation 3 bounds Equation 2 to a maximum value, the wanted exection time E. Equation 4 implies for only positive integers as values for l_i. To add the initial cost value only once and when the benchmark program is selected ($l_i > 0$), y_i can only be 0 or 1 depending the value of l_i. This is realized by Equations 5 en 6. The output of the ILP is for each benchmark program the value of l; if l is bigger than zero, the minimum number of executions for that benchmark program is added to l and the benchmark program is selected. The tool uses the GNU Linear Programming Kit (GLPK) [6] to calculate the ILP. The output of this part is an XML file per task set, containing the selected benchmark programs and the number of executions of each program. Based on the selected benchmark programs and the number of executions, the source code is generated.

4.3 Generating Executable Tasks

The final part of the tool-chain generates the source code for each task and a *makefile* for every task set. It uses the task sets from Section 4.2 as input. For each task of a task set the initialization and main function of all selected benchmarks are called from within the code of the task. Afterwards a makefile is generated that compiles the tasks for the target platform. Each benchmark program is called within a *for-loop* statement, where l is the loop bound. After the code of the benchmark programs is appended, additional lines of code are inserted at the beginning and the end of the code, depending on the user requirements. This header and footer code can be added and/or changed in the template file. Listing 1 shows an example of generated source code. Compiling the source code files using the makefile, results in a set of executables which can be executed on the target hardware.

Listing 1: Example of generated code of a Task with minimal header and footer code

```
//header_begin
int task(void){
//header_end
    int i;
    for(i = 0; i <606; i++){
        bitonic_init();
        bitonic_main();
    }
    for(i = 0; i <103; i++){
        h264dec_init();
        h264dec_main();
    }
//footer_begin
}
//footer_end
```

In this section we have introduced a task-set generator tool that can be used to generate tasks for the formal proof, analysis by simulation and for the analysis on the implementation level. In the next section, we will evaluate the generated executable tasks by executing them on the target

platform and by comparing measured and targeted execution times of the tasks.

5. EXPERIMENTAL EVALUATION

In this section, we report on the experiments using the task-set generator and the results of these experiments. To compare the performance of the scheduling algorithm on different evaluation levels, the utilization of the generated task sets must be identical on each level. Consequently, the execution time of each task must be equal to the targeted WCET of the task as used in the the first two evaluation levels, or at least within an acceptable margin. The aim of our experiment is to examine the deviations of the task behaviour at the three evaluation levels: (i) the formal analysis, (ii) the simulation based analysis, and (iii) the analysis at the implementation level.

We have generated a number of synthetic task sets and executables using the task-set generator. To create the executables, we first need to derive the timing behaviour of the benchmark programs. To this end, we have executed and measured 10 benchmark programs to obtain their execution times and their minimum required number of executions. After updating the description files, the task-set generator calculates the benchmark program sequence of each task.

5.1 Experiment Setup

We have conducted our experiments on a platform with an Intel(R) Xeon(R) CPU E5-2420 v2 processors at 2.20 GHz. Xen 4.5 was patched with the latest version of RT-Xen[2]. The guest domain was installed with a para-virtualized kernel. Dom0 is booted with two VCPUs, each pinned on a PCPU, and 4GB memory. The remaining ten cores where used to run the guest domain. The scheduling algorithm of the guest OS, patched by LITMUS^{RT}, is a global Earliest Deadline First (EDF) algorithm. In our experiments tasks are generated based on the *base_task* from the LITMUS^{RT} library. For tracing the tasks in the *feather-trace* tool, included by LITMUS^{RT}, was used.

5.2 Execution Cycles of the Benchmarks

We selected 10 benchmark programs from the TACLeBench benchmark suite. Before these programs can be used to create executable tasks, the execution time of the benchmark must be known. To create reproducible task times, a minimum number of executions is required for each benchmark program (Section 4.2). For this, we analyse the execution time of a benchmark program by executing the benchmarks a statistically relevant number of times l. Typically for each benchmark we do $l = \{10, 50, 100, 500, 1000, 2000\}$ measurements. Besides the execution times, we calculate the mean execution time T. Based on the above measurements a value l' exists: $\forall l > l' : l \cdot T \approx T_l$ with T_l denoting the execution time of running the benchmark l times. This value l' is defined as the minimum required executions for the benchmark program. This results in an execution time T and a minimum required executions l' for each benchmark program. We analysed the execution of each benchmark program, and observed that the execution time has a standard deviation of less than 1% if the benchmark program is executed for at least l' times. To correct for the fluctuation of a task's execution time, an extra safety margin M has to be added to the number of execution cycles of the benchmark programs. We have found that a safety margin $M = 2\%$ suffices to ensure that the execution time does not exceed the WCET of the task. The calculated execution times and the minimum required executions for the corresponding architecture (in this case x86) are added to the description file of the benchmark program. See Figure 2.

[2]https://sites.google.com/site/realtimexen/

```
<program name="ndes" path="bench/sequential/ndes">
    <parameter name="WCET" var0="x86" var1="31518"/>
    <parameter name="min_n" var0="200"/>
</program>
```

Figure 2: Example of a benchmark program description file

5.3 Creating and Executing Tasks

After updating the description files, we generated the synthetic task sets and the executable tasks. We generated 5 randomized synthetic task sets $S_1, ..., S_5$, each with 20 tasks. For each task we executed the ILP program and generated the source code based on the benchmark sequence of the task. For our experiment, we have compiled the tasks as shared libraries to call the task function (Listing 1) in each job of the real-time task in LITMUS^{RT}. Because the focus of the experiment lies on measuring the execution time of tasks, we used one guest with one dedicated core and pinned the virtual cpu to a physical cpu and executed the tasks one by one. The runtime of the experiment per task is 10 seconds and we repeat the experiment 10 times. Since the aim of the experiment is deriving the execution times of the tasks, the period of each task is fixed to 1 second, this gives us an equal number of measurements for each task. This results in 100 measurements for every task of the 5 task sets.

5.4 Results

The results are shown in Figure 3. The experiments shows that the calculated WCET of the ILP program of each task has a deviation of less than 0.00001% compared the target WCET generated in the first part of the task-set generator. Comparing the execution time of the tasks on the target platform to the target WCET demonstrates that the execution time does not exceed the target WCET of the tasks. Secondly, the task with the lowest minimum execution time is task τ_9 of task set S_5 (see Figure 3) with an execution time of 96.4% of the target WCET. Decreasing the safety margin M would result in an higher lower bound of the execution time. The upper bound of the execution time, however, would exceed the WCET and could result in an overload situation.

6. CONCLUSION AND FUTURE WORK

This paper presents a next generation task-set generator tool. It generates reproducible task sets for the three evaluation levels to evaluate and compare scheduling algorithms for state of the art technologies. For the first two levels, synthetic task sets are generated that do not bias against any scheduling algorithm. Pseudo-random generated values make it possible to generate reproducible task sets. For the third level, a new task set generator method to create executable tasks for measurement-based analysis has been introduced. Using publicly available benchmark programs, making test sets reproducible and also making this tool open

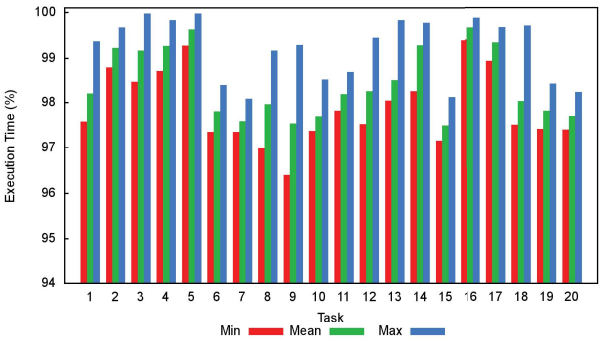

Figure 3: Task Set S_5

source enables the possibility to standardize the proposed methodology. We have evaluated the task-set generator tool in a number of experiments. We establish that the execution time of the tasks do not exceed the targeted WCET of the task, and has a lower bound of 96.4% of the WCET. The development of the task-set generator tool is a ongoing process. At this moment a first version of this tool is publicly available under the GPL license. This version of the tool can be used to generate task sets with executable tasks for the x86 architecture using 10 benchmark programs.

As this is the first version of a task-set generator, there is room for many improvements and extensions. We will focus on the two topics with the highest priority:

1. Synthetic task-set generator:
   - Extend the number of supported task models. The concept is that users should be able to tune the input parameters in a sense that the generated task sets support their research goals.
   - Extend the number of supported task set generator algorithms. The intention is to create a framework in which current but also future algorithms can be plugged in.
2. Benchmark selection:
   - Improve the method to create a stable execution time due to cache effects. This would result in more stable execution times, and would decrease the safety margin M while not exceeding the targeted WCETs of the tasks.
   - Increase the number of processor architectures. This would give us and other researchers the possibility to evaluate and compare scheduling algorithms on different target platforms using identical synthetic task sets.
   - Make the selection of the benchmark programs not only based on the architecture of the target platform, but also on other criteria. After preselecting the benchmark programs with support of the chosen architecture, the user will be able to select an adjusted set of benchmarks. Possible selection criteria are the use of floating point units, the size of the benchmarks, or the domain of the benchmarks.

7. ACKNOWLEDGMENT

This study was funded by the Agency for Innovation by Science and Technology in Flanders (IWT).

8. REFERENCES

[1] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In *Formal Modeling and Analysis of Timed Systems (FORMATS)*. Springer Berlin Heidelberg, 2003.

[2] R. I. Davis and A. Burns. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. *ACM Computing Surveys*, 1(4), 2009.

[3] P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2010.

[4] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, R. B. S. Schoeberl, P. Waegemann, and S. Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, 2016.

[5] S. Kramer, D. Ziegenbein, and A. Hamann. Real World Automotive Benchmarks For Free. In *the 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[6] A. Makhorin. GNU Linear Programming Kit (GLPK), 2012.

[7] J. P. Vielma. Mixed Integer Linear Programming Formulation Techniques. *Society for Industrial and Applied Mathematics (SIAM)*, 57(1), 2015.

[8] P. Wägemann, T. Distler, T. Hönig, V. Sieh, and W. Schröder-preikschat. GenE : A benchmark generator for WCET analysis. *Open Access Series in Informatics (OASIcs)*, 47, 2015.