Arbeit zur Erlangung des akademischen Grades
Master of Science

Generating Real-Time System Specifications with Known Response Times

Henning Brockmann

2025

Lehrstuhl für Informatik 12
Fakultät Informatik
Technische Universität Dortmund

Erstgutachter:	Prof. Dr. Peter Ulbrich
Zweitgutachter:	Alwin Berger
Abgabedatum:	03.März 2025

Abstract

Modern real-time systems face significant challenges in ensuring that tasks meet their deadlines, primarily due to the unpredictability of worst-case execution time (WCET). Accurate timing analysis is crucial for the validation of these systems, but it is often hindered by the complexity and variability of real-world applications. Benchmarks play a vital role in this context, providing a standardized way to evaluate and validate timing analysis techniques. This thesis explores the development of a generator for benchmarks, aimed at creating diverse and representative test cases that can be used to rigorously assess the performance and reliability of timing analysis methods in real-time systems.

Contents

1 Introduction

1.1 Background and Context

In current times, embedded systems are more ubiquitous and are used in a wide range of applications. Mostly, but not solely, known for their use in the automotive industry, many people use embedded systems daily. Since the uses are often in situations where the system must react in real-time, the systems must be designed with high reliability and efficiency. For many systems promptly decision making and immediate reaction is crucial, like in the automotive industry, where for example the break control or airbag system must react in a fraction of a second to safe lives. Staying at the example of the automotive industry, the use of embedded systems is not only limited to the control of the engine or the transmission, but also includes the control of the infotainment system, the climate control, and the driver assistance systems. To ensure that the system can react in real-time, the system has to be designed to handle the required tasks on time. Since many variables of the circumstances are inconsistent, but may be crucial for a system to react in time, the system must be designed to handle the worst-case scenario. This also means that the system must be designed to handle the longest time a task needs to execute on the given system, also known as the WCET. With the WCET not known, it is the goal of analysis tools to bound the WCET as precise as possible while not over- or underestimating the time too much. Determining the WCET is a complex task in different ways. One way is the use of static code analysis, which is done by analyzing the system's source code and deriving the needed execution time[4]. Another way is dynamic and measurement based analysis, which is done by executing and tracing the functions of the system deriving the time that the system needed to execute the task[4]. Either way, the analysis has to be checked for correctness, and the results must be validated. To do this, the analysis methods are tested against a set of benchmarks, standardized pieces of software to portrait different software designs and patterns, like recursion, loops, memory access, and so on. Some collections of benchmark software like the Mälardalen benchmark[14] or the TACLeBench[13] are used to present a broad collection of benchmarks to test the analysis tools. But even with the benchmark collections, there exist some restrictions like limitations in operating systems, compilers and system configurations.[13, 14]

1.2 Problem Statement

Existing tools often do lack some parts of occurring problems in real life. For most parts of the analysis, the benchmarks do not cover all possible scenarios and system configurations. Gustafsson et al. [14] listed some weak-points to be improved in the Mälardalen benchmarks. The size of the benchmarks is one aspect Gustafsson et al. see as a point of interest. While the smaller sized benchmarks are fine to check for a specific type of programming construct, they do not show how tools will handle with bigger programs. Due to the size and lack of complexity the Mälardalen benchmarks can not compare with real-time industrial applications. With few exceptions the algorithms and programs to not represent the code size and code constructs, that would be used in an industrial use case. Further is explained the focus relies mostly on flow analysis while the influence of e.g. caches, data caches and other hardware effects is not depicted by the benchmarks. While not depicting hardware effects, the benchmarks of the Mälardalen project are only written in C.

With the Mälardalen benchmark project as example it shows a broad range of aspects benchmarks ideally represent.

Other researchers have chosen to tackle the problem by building generators that create systems presenting interesting problems with known response times. These generators are designed to produce synthetic task sets trying to mimic the behavior and complexity of real-world applications. By controlling the parameters and characteristics of the generated tasks, researchers can create benchmarks that are tailored to specific analysis needs. Some well known generators are TASKers[12], *Sys*WCET[10] or GenEE[11].

These approaches highlight the importance of having flexible and configurable generators that can produce diverse and representative task sets for benchmarking purposes.

1.3 Research Questions and Objective

This thesis aims to address several key research questions related to the generation of task sets with known timing characteristics:

1. How can task sets with known WCET and worst-case response time (WCRT) be generated to accurately reflect the complexity of real-world systems?

2. What methods can be employed to ensure that the generated task sets are flexible and diverse, allowing for variations in system configurations, operating systems, and hardware platforms?
3. How can the correctness and reliability of the generated task sets be validated in terms of their timing characteristics?
4. How can the generated task sets be utilized to improve the accuracy and efficiency of WCET and WCRT analysis methods?

By addressing these questions, this thesis aims to develop a framework for generating task sets with known WCET and WCRT, providing a valuable resource for the evaluation and improvement of timing analysis tools. Besides that the generator shall create an opportunity for flexible variations in generating different systems by not limiting itself to a single target compiler, but targeting a meta-model of a system to later on allow for diverse porting to different compilers and operating systems.

Since in real-life situations it is near impossible to derive all possible scenarios and branches of a piece of software In order to overcome known problems the plan is to create the resulting system configuration bottom up. This approach is intended to allow for a more flexible and diverse generation of task sets that can better represent the complexity of real-world systems.

1.4 Structure of the Thesis

This thesis is structured into several chapters, each building upon the previous to provide a comprehensive understanding of the research conducted.

Chapter 2 delves into the fundamental concepts and literature review related to real-time systems, WCET and WCRT analysis, and benchmarking. This chapter introduces essential concepts and theories that form the basis for the subsequent chapters.

In **??** the proposed approach for *Generating Real-Time System Specifications with Known Response Times* is outlined. This chapter details the methods and techniques employed in the generation process, providing a clear understanding of the research methodology.

The Chapter 4 provides an in-depth look at the implementation of the proposed approach. It describes the practical aspects of the research, including the tools, technologies, and processes used to bring the theoretical concepts to life.

Finally **??**, concludes the thesis with a summary of the findings and a discussion of their implications for the research. This chapter reflects on the research questions and objectives, evaluates the success of the proposed approach, and suggests areas for future work.

2 Basics

This chapter provides a collection of basic knowledge as a foundation for this work.

2.1 Real Time Operating Systems

A Real-Time Operating System (RTOS) is designed with precise timing and high reliability in mind for running applications[26]. Such systems are used in a wide range of applications, from multimedia systems and smart home systems to automotive and medical sectors, such as pacemakers[15]. In all these systems, the correctness and timeliness of the applications are of utmost importance[15].

2.1.1 Tasks and Jobs

According to the *IEEE Standard for a Real-Time Operating System (RTOS) for Small-Scale Embedded Systems* [18], a task is defined as the "basic logical unit of concurrent program execution" that contains code. The standard also states that while the code within a task is executed sequentially, the tasks themselves may run in parallel with other tasks.

Tasks can have deadlines based on their importance and the nature of their work. If a task misses its deadline due to insufficient computational time, the system and results may be affected differently. It is common to categorize task deadlines into three types[2, 9, 25]:

1. **Hard Deadline:** Missing a hard deadline is considered a system failure. The task must complete by its deadline; otherwise, the system may enter an unsafe state.
2. **Firm Deadline:** Missing a firm deadline does not cause a system failure, but the task's result is no longer useful and is discarded. The system continues to operate, but the quality of service may degrade.
3. **Soft Deadline:** Missing a soft deadline reduces the task's utility, but the result can still be used. The system can tolerate some deadline misses without significant impact on overall performance.

A job is the specific execution instance of a given task when scheduled. One task may be assigned with multiple jobs. If a task is scheduled to execute every x time units, the total number of jobs will be equal to the number of times x fits within the inspection cycle. For a more comprehensive description on tasks and jobs, refer to Section 3.1.1.

2.1.2 Real-time Scheduling

Since running a single Task would not necessarily require a scheduler, running multiple tasks in a way to satisfy the requirements a real-time system, especially a hard real-time system presents is topic of a wide field of research.

Creating the schedule of given tasks to be executed some aspects

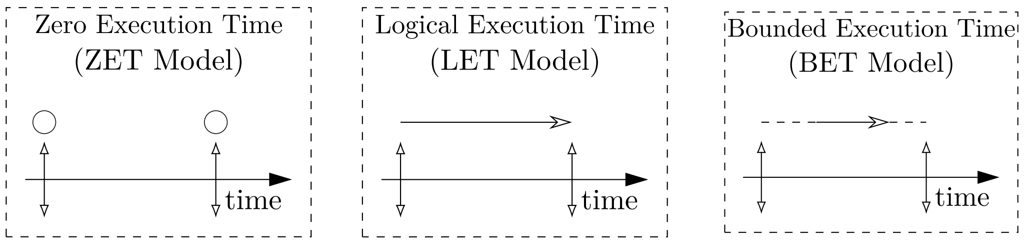
Generating a task set with known WCET and WCRT requires knowledge about the schedule. Implementing a scheduler gives the benefit of taking direct insights in the compilation of the generated results. To determine the order of operation when being confronted with two or more tasks, the tasks need to get a priority assigned.

There are some scheduling techniques to determine the priority of tasks to be scheduled.

- **Clock based Scheduling:** With clock based scheduling every task gets a share of execution time in which the work of the task may be done. It is a way to schedule tasks without priorities whilst making sure every task gets time on the computational unit. While being a simple algorithm it does create a lot of overhead, since the timeslots do not necessarily encompass the whole time needed by the task.
- **(Weighted) Round Robin:** In basic Round Robin scheduling, each task is assigned a fixed time slice or quantum in a cyclic order, ensuring all tasks get an equal share of CPU time. Weighted Round Robin extends this by assigning different weights to tasks, allowing tasks with higher weights to receive more CPU time compared to tasks with lower weights. This provides a more flexible and efficient scheduling mechanism, especially for systems with tasks of varying importance or computational needs. [16]
- **Rate Monotonic Scheduling:** Rate monotonic scheduling is a fixed priority algorithm where tasks with shorter periods are assigned higher priorities. It is optimal for fixed priority scheduling under certain conditions. [21]
- **Earliest Deadline First:** With the earliest deadline first scheduling algorithm, the priority of the tasks is not fixed, but dynamic. As the name implies, the tasks get a priority assigned in a way to ensure the tasks with the nearest upcoming deadline get to run first. [22]

2.1.3 Execution Time Paradigms

In the calculation of WCET the reading of input and the writing of output often led to too pessimistic estimations of the needed execution time. Fig. 2.1 displays



(a) Zero Execution Time **(b)** Logical Execution Time **(c)** Bounded Execution Time

Figure 2.1: Display of three different programming abstractions[5].

three different approaches to create an abstraction of timing models: Zero Execution Time (ZET), Logical Execution Time (LET) and Bounded Execution Time (BET). With the approach of ZET the assumption is to completely ignore the timing aspects of reading and writing output and just assume instantiations effects[5]. LET started as a programming model, but led to a well-known paradigm for analyzing real-time systems to decouple the timing behavior of tasks from the actual execution on the hardware. In the LET model, the execution time of a task is defined logically rather than physically. This means that a task is considered to start and finish at predefined logical times, regardless of when it actually starts and finishes on the hardware[5]. The BET approach takes a step away from the abstraction of execution times by bounding them in time frames using deadlines marking upper bounds in which the interaction has to be finished[5].

The concept LET that developed from the programming language Giotto[17]. In their paper Henzinger, Horowitz, and Kirsch [17] present several benefits of the LET model: Firstly, it ensures predictability by defining the execution times logically. This predictability is crucial for real-time systems as it guarantees that the timing behavior of tasks remains consistent and reliable.

Secondly, the LET model allows for deterministic behavior. Since the logical execution times are fixed and do not depend on the actual execution times, the system's behavior remains deterministic, which is essential for safety-critical applications.

Lastly, the LET model decouples the timing behavior of tasks from the underlying hardware. This decoupling makes it easier to port the system to different hardware platforms, enhancing the system's flexibility and adaptability.

The LET model is commonly used in safety-critical systems, such as automotive and aerospace applications, where predictability and determinism are essential.

2.2 Timing Analysis

For real-time systems it is desired to keep up the reliability and predictability of the given tasks to ensure their timely execution respectively to their deadlines. To deduce a task's worst execution time without underestimating, WCET analysis is employed. Various approaches are currently used to determine the possible execution times of tasks, including static analysis, measurement-based analysis, and probabilistic analysis, which will be discussed in the upcoming Sections 2.2.1, 2.2.3 and 2.2.4.

2.2.1 Static Analysis

Static analysis makes use of several techniques to generate a model of the given hardware and software to gain knowledge about the executional behavior of the inspected code. One key benefit of the static analysis is that the code does not need to be executed.

Wilhelm et al. [27] describe the static analysis in multiple phases. These phases are *control-flow analysis*, *processor behavior analysis* and *path analysis*.

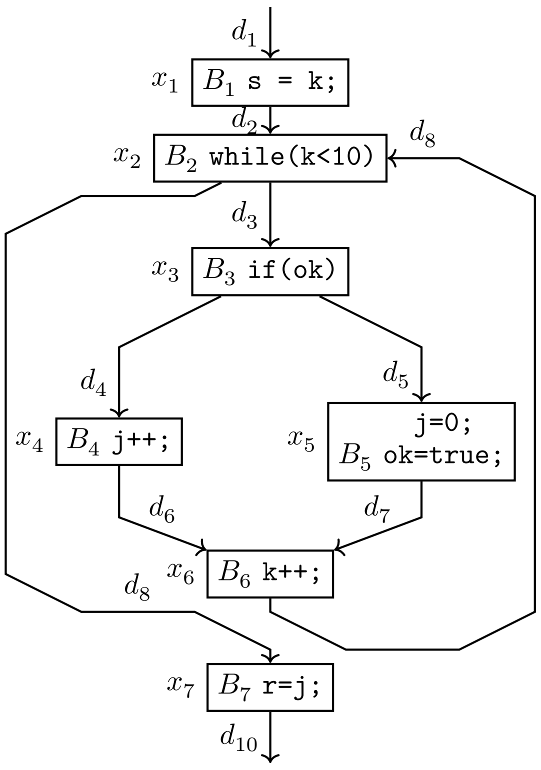
Control-Flow Analysis

When analyzing a given task, a CFG is built to represent the execution paths of the task's code, including a task's call graph. The control-flow analysis, previously called *high-level analysis*, attempts to gain as much knowledge about the control flow of a given task by identifying loops and their bounds, flow branches, and function calls. When including information about the data flow of the task, it may be possible to exclude paths from the CFG that are infeasible and may not be executed at all, for example, paths with mutually exclusive conditions.

Figure 2.2 illustrates an example code snippet and its corresponding CFG. The code snippet, shown in Fig. 2.2a, contains loops and branches that determine the sequence of operations based on the value of the variable `k`. The CFG, depicted in Fig. 2.2b, visually represents the basic blocks of the piece of code, highlighting the possible execution paths. On base of the CFG and some additional information, like the loop bound of $k < 10$ in this example, it is possible to deduct constraints

```
/* k>= 0 */
s = k;
while (k < 10){
        if(ok)
                j++
        else{
                j = 0;
                ok = true;
        }
        k++;
}
r = j;
```

(a) Example code



(b) control-flow graph (CFG) for Fig. 2.2a

Figure 2.2: Example code of an loop with branching code and its representation as a CFG (figure taken from [23]).

describing the flow of the code enabling an estimation of the costs of different execution paths through the graph[23, 27]. This technique is called implicit-path enumeration (IPET)[27].

2.2.2 Data Flow Analysis

data flow analysis (DFA) is used to analyse the flow of data in a program to detect dependencies and bottlenecks. Some bottlenecks may be cache misses, e.g., because an Least Recently Used (LRU) entry got overwritten. Cache misses can significantly impact the execution time of a task, as accessing data from the main memory is much slower compared to accessing it from the cache[23]. This can lead to increased execution times and variability in the timing behavior of tasks. For this reason it is of interest to depict the cache's status to know if the data access results in a cache miss or a cache hit.

timing accidents - Chen 04 10/53

Static analysis is advantageous because it does not require actual execution of the code, making it suitable for early stages of development. However, it can be overly conservative, leading to pessimistic WCET estimates due to the need to account for all possible execution paths and hardware behaviors.

2.2.3 Measurement-based Analysis

Measurement-based analysis involves executing the code on the target hardware and measuring the execution times to determine the **wcet!** (**wcet!**). This approach relies on collecting empirical data from actual runs of the system, which can provide more accurate and realistic estimates of execution times compared to purely static methods. By running the code under various conditions and workloads, measurement-based analysis can capture the dynamic behavior of the system, including the effects of caches, pipelines, and other hardware features that may impact execution time. However, this method requires a comprehensive set of test cases to ensure that all possible execution paths are covered, which can be challenging and time-consuming[27].

One of the main advantages of measurement-based analysis is its ability to provide precise timing information that reflects the actual performance of the system. This is particularly useful for complex systems where static analysis may be overly conservative and result in pessimistic WCET estimates. However, measurement-based analysis also has limitations, such as the difficulty in guaranteeing that the worst-case scenario has been observed during testing. Additionally, this approach may not be suitable for early stages of development when the hardware is not yet available, in which case appropriate simulators are used trying to mimic real hardware.[27] To address these challenges, hybrid approaches that combine measurement-based and static analysis techniques are often employed to achieve more accurate and reliable WCET estimates[19].

2.2.4 Probabilistic Analysis

Early flowcharts and Abstract Interpretation (AI)s described by Cousot and Cousot [7]

cache misses

overestimation & pessimism

Kelter [19] mentions several techniques that can be used for WCET analysis, including static analysis, measurement-based analysis, statistical analysis and hybrid approaches. Static analysis involves analyzing the code without executing it, while

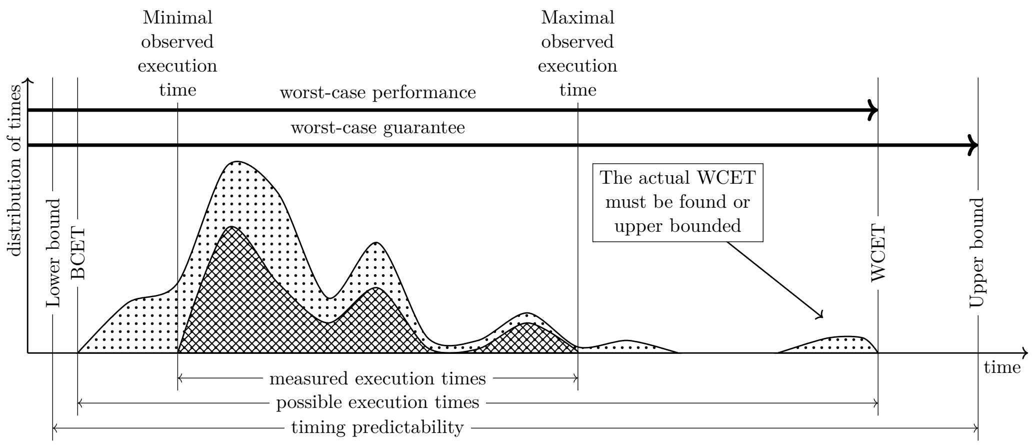


Figure 2.3: The graph above illustrates two curves representing the basic concepts of timing analysis. The first curve, with the dotted area, shows the range of all possible execution times, with the minimum and maximum values being the best-case execution time (BCET) and WCET, respectively. The second curve, with the crosshatched area, represents the range of execution times identified through analysis, with the minimum and maximum values being the *minimal observed execution time* and *maximal observed execution time*. This graph is adapted from Wilhelm et al. [27].

measurement-based analysis involves running the code on the target hardware and measuring the execution times. Hybrid approaches combine both static and measurement-based techniques to provide more accurate WCET estimates.

Static analysis is a proper and safe way to get information about the execution times, since it is unhinged from hardware constraints and derived from a meta model created from reviewing the code [19].

To evaluate analysis techniques benchmarks play a significant role. Providing multiple task sets with known execution scenarios and the tasks behavior the analysis techniques can be tested for its effectiveness. Benchmarks commonly used are for example the Mälardalen WCET benchmark and the TACLeBench benchmark suite[13].

Accurate WCET analysis is critical for the design and verification of real-time systems, ensuring that all tasks can be scheduled and executed within their deadlines, thereby maintaining the system's overall reliability and performance[19]. Kelter [19] states further that WCET analyses are no longer sufficient for newer and complex systems and need to analyze delays and preemptions between tasks as well, resulting in WCRT.

2.2.5 Critical Instant Theorem

The Critical Instant Theorem, introduced by Liu and Layland [24], states that the WCRT of a task occurs when it is released simultaneously with all higher-priority tasks. This theorem is fundamental in fixed-priority scheduling, as it helps in determining the worst-case scenario for task execution.

According to the theorem, to find the worst-case response time of a task, one must consider the scenario where the task is released at the same time as all higher-priority tasks. This situation is known as the critical instant. By analyzing the system under this condition, one can ensure that the calculated response times are indeed the worst-case values.

The theorem is particularly useful in the context of Rate Monotonic Scheduling (RMS) and other fixed-priority scheduling algorithms, as it provides a systematic way to evaluate the schedulability of tasks. If all tasks meet their deadlines under the critical instant scenario, the system is considered schedulable.

2.2.6 Related Work

[8] Built taskgraph with harmonic set

3 Concept

3.1 System-model

In this section, we define the system model that serves as the foundation for the concept of the generated task-sets. It is reduced to a description of the task model and the execution model without further defining a hardware model, since the generated model is designed to be a meta model with no constraints on hardware specifications, making it hardware independent.

3.1.1 Task Model

As a foundation for this generator, the following section defines the bounds and properties of tasks, jobs, and sets of tasks. This builds upon the basic concepts introduced in Section 2.1.1, providing a more formal description with additional properties and information.

Tasks

Each task τ_i may be described by multiple points of time:

- a_i: time of arrival
- r_i: release time
- d_i: deadline.

In this model, tasks are periodic with implicit deadlines, meaning that the arrival time a_i, release time r_i, and deadline d_i can all be described by a single period value T_i. Specifically, the arrival time a_i is the start of the period, the release time r_i is the same as the arrival time, and the deadline d_i is the end of the period. This simplifies the task model as follows:

- $a_i = k \cdot T_i$
- $r_i = a_i$

- $d_i = (k+1) \cdot T_i$

where k is a non-negative integer representing the k-th instance of the task. This restriction is applied for the same reason mentioned by Dar-Tzen Peng, Shin, and Abdelzaher [8], as the allocation of aperiodic tasks can be considered "a dynamic load sharing problem"[8]. This means that the system must dynamically allocate resources to tasks as they arrive, which can lead to unpredictable behavior and potential overload if not managed properly. By focusing on periodic tasks with well-defined periods and deadlines, the system can ensure a more predictable and manageable load, improving overall schedulability and reliability.

Given that the proposed generator leverages detailed knowledge about the execution times of task jobs, additional details are provided to describe it. Each task is assigned multiple execution times, representing the possible outcomes depending on the system's status when scheduled. This includes adding a list of possible execution times. While $e_{i,k}$ defines the actual execution time of a job $\tau_{i,k}$[4], let's define e_{i,k^poss} as a *possible* runtime of the job. This will be described in more detail in Section 3.1.1.

Task-set

Two or more tasks would be considered a task-set. In this model, a task-set Γ is simply represented as a set of all given n tasks $\Gamma := \{\tau_1, \tau_2, \dots, \tau_n\}$.

To provide some interaction and influence between multiple tasks a chain Γ^c is presented. The model of a chain is widely described in research[1, 3, 6]. A chain Γ^c describes an *ordered* list of tasks τ_1 to τ_m.

$$\Gamma^c := [\tau_s, \tau_{m_1}, \tau_{m_2}, \dots, \tau_e]$$

- τ_s: the first task of the chain
- τ_{m_*}: intermediate task of the chain
- τ_e: the last task of the chain

Each element of that list is denoted with its index as $\Gamma^c[i]$ where $\Gamma^c[1]$ represents the first list element τ_s[6]. The elements in the list are ordered so the influence of the tasks is structured as $\tau_{m_x} \to \tau_{m_{x+1}}$. A task $\tau_{m_{x+1}}$ is dependent on the outcome of τ_{m_x}. The interaction between two tasks is done via shared memory on the premise the value is read at the start of a job and written at the end of a job[6]. This corresponds to the model described in Section 2.1.3 of LET.

Each task set generated by this scheduler is designed to be a harmonic task-set, meaning that the periods of all tasks are multiples of each other[20, 24]. As Liu and Layland [24] state it will allow the system to reach an utilization of up to 1, whereas any non-harmonic set of task, when using a fixed-priority rate monotonic scheduling algorithm, will be bound in its utilization.

Jobs

A job $\tau_{i,k}$ represents a single instance of a task τ_i (see Section 2.1.1) and is characterized by the following properties:

- k: index of the job $\tau_{i,k}$ with $k\epsilon[0,\dots,\mathit{hyperperiod}/\max(T_*)]$
- $r_{i,k}$: release time of job $\tau_{i,k}$
- $d_{i,k}$: deadline of job $\tau_{i,k}$
- $e_{i,k}$: computational time of job $\tau_{i,k}$

Due to being a harmonic task-set and its properties (see Section 2.1.2), the *planning cycle*[8] of $\mathit{\Gamma}$ is defined by the *least common multiple (LCM)* of all tasks' periods $\{T_1,\dots,T_n\}$, resulting in the period T_{max} of the task with the highest period.

As in Section 3.1.1 presented the possible execution times e_{i,k^poss} describe possible execution times a job $\tau_{i,k}$ may be assigned. In Section 3.2.5 explains the execution times' generation in more detail.

Each task τ_i is characterized by its period T_i, its list of possible execution times $\{e_{i,1_poss}, e_{i,2_poss}, \dots\}$, and the set of related jobs $\tau_i = \{\tau_{i_0}, \tau_{i_1}, \dots, \tau_{i_m}; m = \frac{P_{max}}{P_i} - 1\}$.

Each job τ_{i_j} is described by the release time r_{i_m}, the beginning of the next period $P_i + 1$ is equal to the implicit deadline d_i, and the job's computational time C_{i_m}. The release time r_{i_m} per job is set to be aligned with the period beginning with 0.

$$r_{i_m} = P_i \times m$$
$$d_{i_m} = P_i + r_{i_m}$$

explain chains and their length

3.1.2 Execution Model

As already mentioned in Section 3.1.1 the employed scheduler will be a fixed-priority rate-monotonic scheduler (see Section 2.1.2). The scheduler operates by assigning each task T_i a priority inversely proportional to its period P_i **??**. Tasks with shorter periods receive higher priorities and are scheduled more frequently. Higher-priority tasks can preempt lower-priority tasks. If a higher-priority task becomes ready to execute while a lower-priority task is running, the scheduler will interrupt the lower-priority task and allocate the processor to the higher-priority task. This is represented in the resulting schedule as jobs being divided into multiple segments $C_{i,k}^1$ to $C_{i,k}^j$ where j is equal to the number of preemptions plus 1. The resulting schedule is a list of $C_{i,k}^j$ depicting the complete schedule. Each $C_{i,k}^j$ has its own starting time $s_{i,k}^j$ and its finishing time $s_{i,k}^j$.

Making use of the RMS in combination with the in Section 3.1.1 mentioned harmonic task-set it is possible to reach utilizations up to 1 [24].

Buttazzo [4, p. 70f] explains some assumptions to consider in the scheduling:

A1 The instances of a periodic task τ_i are regular

A2 . The instances of a periodic task τ_i are regularly activated at a constant rate. The interval T_i between two consecutive activations is the period of the task.

A3 . All instances of a periodic task τ_i have the same worst-case execution time C_i.

A4 . All instances of a periodic task τ_i have the same relative deadline d_i, which is equal to the period $.T_i$.

A5 . All tasks in T are independent; that is, there are no precedence relations and no resource constraints.

A6 . No task can suspend itself, for example, on I/O operations.

A7 . All tasks are released as soon as they arrive.

A8 . All overheads in the kernel are assumed to be zero.

buttazzoHardRealTimeComputing2024 explains scheduling with some assumptions

[8]

3.2 Concept

The general workflow involves several key steps.

1. Generate period times based on the server model.

   what is a server model?

2. Create task chains and identify any missing links within these chains.

   what is a chain?

3. Distribute the load across each period and generate tasks with lower execution times to balance the workload.
4. Connect tasks and create conditions, considering the idea of workloads with resources attached, such as inter-task communication.

   what is a connection?

5. Spread the server model into multiple server instances.

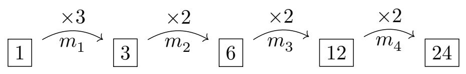
   server models / server instances. what is this?

6. Build a scheduler and perform a schedulability test using the tree structure of the previously built server instances.
7. Export the generated task-set for further use.

3.2.1 Generating Period Times

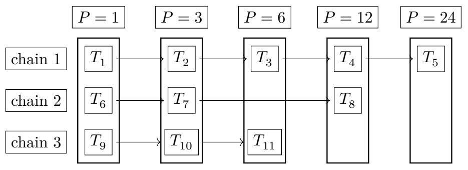
For the purpose of keeping the generated task-set harmonic, the periods to which the tasks will be assigned are generated first. The period being an abstract value in this scenario, the starting value in the generation is selected to be equal to 1. Every further period is selected by randomly choosing an integer value to multiply the current value with. The multiplication defining the amount of jobs being generated, it may be favorable to keep this randomly chosen integer to be relatively small whilst being at minimum the value of 2. This is done until a previously defined amount of values is generated, limiting the maximum length of the inter-task communication chains.

With the multiplication of the previous value a harmonic task-set is assured, since the LCM is per definition the biggest generated period value. In Section 3.2.1 the maximum value is 24 being a multiple of all lower periods. Hence the planning cycle matches with the highest period of the task-set, in the given example the resulting time frame used to create the model is $[0, 24[$.

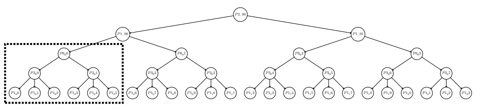


$$\begin{tikzcd}\boxed{1} \arrow[r, bend left, "\times 3", "m_1"'] & \boxed{3} \arrow[r, bend left, "\times 2", "m_2"'] & \boxed{6} \arrow[r, bend left, "\times 2", "m_3"'] & \boxed{12} \arrow[r, bend left, "\times 2", "m_4"'] & \boxed{24}\end{tikzcd}$$

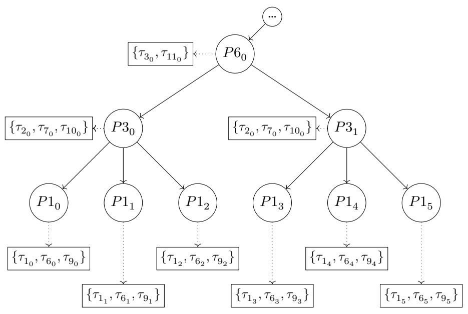
(a) Generation of a small example series of periods by multiplication



(b) Task-set $T = \{T_1, \dots, T_{11}\}$ parted into three subsets each a representation of a chain of inter-task communication



(c) A graph depicting the generated periods. Highlighted with a dotted box: subtree with $P6_0$ as root (see **??**).



...
$P6_0$
$\{\tau_{3_0}, \tau_{11_0}\}$
$P3_0$
$\{\tau_{2_0}, \tau_{7_0}, \tau_{10_0}\}$
$P3_1$
$\{\tau_{2_0}, \tau_{7_0}, \tau_{10_0}\}$
$P1_0$
$P1_1$
$P1_2$
$P1_3$
$P1_4$
$P1_5$
$\{\tau_{1_0}, \tau_{6_0}, \tau_{9_0}\}$
$\{\tau_{1_1}, \tau_{6_1}, \tau_{9_1}\}$
$\{\tau_{1_2}, \tau_{6_2}, \tau_{9_2}\}$
$\{\tau_{1_3}, \tau_{6_3}, \tau_{9_3}\}$
$\{\tau_{1_4}, \tau_{6_4}, \tau_{9_4}\}$
$\{\tau_{1_5}, \tau_{6_5}, \tau_{9_5}\}$

(d) Subgraph of Section 3.2.1 showing the jobs associated to each representation of period iteration

Figure 3.1: Generation of periods and task-set with tasks with given period

3.2.2 Generating Task Chains

To generate the task-set multiple tasks will be iteratively added and assigned to the already generated periods. By iterating about given period values a natural order of the tasks' periods is assured. With choosing the RMS as scheduling method it is an descending order of priority, since the higher periods will result in lower priorities. Knowing this the tasks can be connected with its successor (if one does exist) to limit the inter-task communication to be directed from higher priorities to lower priorities.

!

priority inversion and resource locking in model

Without Doing so the sender is assured to run earlier in respect to the receiver, if no locking or priority inversion is apparent. By skipping a period while generating the tasks, the chains are built to vary the length and distribution of their associated tasks. The skip may be done at random.

3.2.3 Period-tree

By using a harmonic task-set it is possible to fan out the given list of periods as seen in Section 3.2.1 into a tree structure as seen in Section 3.2.1. Due to the periods being multiples of each other, the used multiple is the key to perfectly map the periods onto the same span of time. Beginning with the smallest period i.e. 1, whilst ignoring the periods greater than that for now, it is a perfect representation of needed planning cycle 1. This is represented by node $P1_0$ in Section 3.2.1. If the next biggest period 3 gets added into the picture, the planning cycle increases to 3, resulting in three repetitions of previous period to match the newer and bigger cycle, represented by $P3_0$ and $P1_0$ to $P1_2$ in Section 3.2.1. Repeating this for the remaining periods, the complete tree, built so far, will be multiplied and added. The benefits of this design will be explained in more detail in Section 3.2.8.

3.2.4 Assigning Computational Time

At this stage, the generated tasks have no computational time assigned. The model is planned to satisfy a given utilization. Taking that utilization, each period may be assigned a combined execution time to represent the sum of associated tasks.

The planning cycle's time frame, multiplied by the predefined utilization, determines the total amount of execution time that can be distributed across the tasks. This means that for each layer of period nodes in the tree structure, the sum of the execution times of all tasks must not exceed the product of the planning cycle's time frame and the predefined utilization.

For example, if the planning cycle's time frame is 24 units and the targeted utilization is 0.75, the total execution time available for all tasks in that planning cycle is $24 \times 0.75 = 18$ units. This 18 units of execution time must be spread across all tasks in the tree, ensuring that the sum of the execution times of tasks associated with each period node does not exceed this limit.

The total execution time for each period layer is then distributed among the tasks assigned to that period. This is done by randomly assigning execution times to each task such that the sum of the execution times equals the total execution time for the period layer.

For example, if a period layer of $P1$ has a total execution time of 6 units and there are three tasks T_1, T_3 and T_9 assigned to that period, the execution times could be randomly assigned as $CP_{1_1} = 1$, $CP_{3_1} = 2$, and $CP_{9_1} = 3$, ensuring that the sum is $CP_{1_1} + CP_{3_1} + CP_{9_1} = 6$ units.

By randomly distributing the execution times in this manner, the system ensures that the overall utilization remains within the desired bounds, maintaining the schedulability and efficiency of the task set.

check utilization use

make sure the times fit. CT<1/2period <- is this correct? is this correct for mult 2 or at all? But spare time is needed, true.

3.2.5 Variation of Execution Times

For the system to not only represent all tasks WCET a variation of execution times per task need to be created. Since the generated tasks with full load still have schedulability, the execution time of the tasks may be reduced without compromising the schedulability. For that reason the execution time of each task is multiplied with a random value between zero and one until a sufficient amount of different values is generated. Keeping the increasing complexity in mind, the system is designed to use a discrete distribution instead of continuous value functions.

value functions more complex than discrete? source

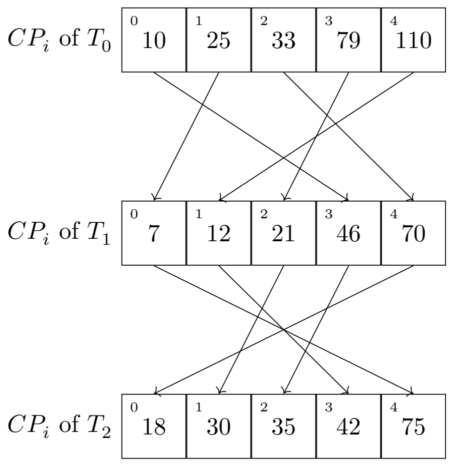
3.2.6 Generating Job Chains

As Section 3.2.2 describes chains of tasks, there is the need to specify these to accommodate for the variations described in Section 3.2.5. Whilst the previously created chains of tasks may describe a general dependency, it is now necessary to use the variations to calculate the actual chains being used in the schedule, as described in Section 3.2.8 later on. For now the creation is not bound to any restrictions and will end in a permutation of the created variations. For example a chain of

3.2.7 Creating Conditions

With multiple possible execution times per task, these variants need to be conditioned to ensure deterministic behavior. This can be achieved by defining conditions that link the execution times of tasks within a chain, ensuring that the execution time of one task influences the execution time of subsequent tasks in the chain, as described in Section 3.2.2. The conditions are supposed to mimic the data-flow between two tasks and the influence between these two. The most simple attempt to combine the different variants of the two tasks together would be by pseudo-randomly assigning a follow-up variant of the second task to each of the first tasks variants. Biggest shortcoming with this idea would the easy predictability of a chain, since the first task would determine the execution times of the following tasks by transitivity. To hide the effects there may be inserted another independent input into the condition.

For example, consider a chain of tasks $\tau_1\{CP : \{10, 25, 33, 79, 110\}\}, \tau_2\{CP : \{7, 12, 21, 46, 70\}\}, \tau_3\{CP : \{18, 30, 35, 42, 75\}$. We can define conditions such that the execution time of τ_1 affects the execution time of τ_2, and the execution time of τ_2 affects the execution time of τ_3. For instance as displayed in Section 3.2.7 the chain has multiple routes to traverse about the possible execution times. For example the first entry in the list of the tasks execution times would lead to the



(a) Unshifted

Figure 3.2: Generation of periods and task-set with tasks with given period

index 3 in second tasks list and finally get to index 2 in the third tasks list resulting in a specific chain $\tau_1 : 10 \rightarrow \tau_2 : 46 \rightarrow \tau_3 : 35$.

By defining such conditions, we ensure that the execution times are linked and the overall behavior of the system remains predictable and deterministic. This approach helps in maintaining the schedulability and efficiency of the task set while accommodating variations in execution times.

3.2.8 Building a Scheduler and Performing Schedulability Test

The schedule can be generated by performing a depth-first traversal of the previously described tree structure. The leaves of the tree represent the tasks with the highest priority and are processed first during the traversal. As the traversal continues, tasks in higher layers with lower priorities are processed until the root is reached. Starting with an empty schedule for the given planning cycle the execution times of the highest prioritized tasks can simply be added. If one layer is added to the schedule, the next layer with the next next higher priority is added into the remaining idle time slots. Since the time and utilization are previously calculated it is ensured the task set is schedulable.

prio for equal periods -> arbitrary, but predictable order.

building a scheduler (schedulability test)

4 Implementation

With the concept described in the previous Chapter 3, the implementation of the generator is described in this chapter.

4.1 Overview

This section provides an overview of the implementation structure and main components of the generator. The implementation is done in Java 21 using the JGraphT library to create the graph described in **??**sec:concept). Java was mainly selected to allow later extensibility with an eye toward a possible integration into the AMALTHEA model, a model for representing real-time systems, using one of their libraries (see Section 5.3).

The generator is designed to flexible and versatile while creating deterministic and repeatable results. For this reason the main application takes commandline arguments referencing configuration files. In the configuration is a given seed value for the random number generator used in the generation process. Every use of random numbers is based on this seed value, which allows for the reproduction of the results whilst keeping the generation process flexible.

4.2 Main Application

The main application is a simple bootstrapping class that initializes the generator, runs the generation process, and outputs the results. For the generation the previous mentioned configuration files are read and used to instantiate the desired input values.

4.3 Graph Builder

The main component of the generator is the graph builder, which designed with a builder pattern in mind. It is employed to build a unidirectional graph without cycles in multiple steps, representing the graph described in Section 3.1.1, especially with Fig. 3.1 in mind. This graph is represented by a class named *TaskGraph*, a wrapping subclass of the *JGraphT* library's DirectedMultigraph.

The graph is built in multiple steps, each changing and morphing the existing graph to result in a final tree containing the complete set of tasks Γ and the associated jobs. The graph builder is designed to be flexible and extensible, allowing for easy modification and extension of the generation process. Each step of the process is represented by a class implementing the *TaskGraphOperation* interface, which is then executed in sequence to build the final graph. The interface is kept simple by taking an existing TaskGraph, running modifications on the given graph and returning the modified graph.

The current implementation includes the following operations described by upcoming Sections 4.4 to 4.7 and **??**.

4.4 Creating harmonic periods

Using an empty graph as input, the operation creates nodes in the graph each representing a period used in the task-set. The initial starting value is set to 1 and all subsequent periods are calculated by multiplying the previous period by a factor f. This factor f is an integer value to ensure the harmonic property of the generated set of tasks (see Section 3.1.1).

The selection of the factor is designed to be a result of a random number generator (RNG), which can be configured in two different ways currently. One way the RNG can be configured it for it to have a mean value and using a Poisson distribution to select the factor. This is an approach to keep the periods in a certain range while allowing outliers and some variation. The other way is to provide a list of possible values with a manual distribution from which the factor randomly is selected. For example, if the list of possible values is $\{2, 2, 2, 2, 2, 2, 2, 3, 3, 4\}$, statistically every tenth every value will be a 4, every fifth value a 3 and the rest will be 2. This approach is designed to allow for a more controlled generation of the periods, keeping the exponential grow of the periods, and thus of the graph being built, at bay. The values and size of the list may be varied to fit the current needs of the generation process.

All generated periods are inserted into the graph as nodes, connected by edges in descending order. The resulting graph is a tree with the root node representing the highest period and the leaf nodes representing the smallest period.

Another configuration option is a global factor for all generated periods to multiply all periods by. This allows for a way to avoid handling floating point numbers in the generation process, which can lead to rounding errors and other issues. For the following a global factor of 1000 is used to keep the periods in the range of milliseconds.

4.5 Assigning computational Time

With the root node of the graph as the highest period, it does represent the hyper-period of the task-set equaling the total amount of computational time available to use, assuming a utilization of 1. This value is then multiplied by a configured value between 0 and 1 to represent the execution time of the tasks. Knowing this amount will satisfy the targeted utilization of the system, the execution time of the tasks can be calculated, while yet now generated, represented by the periods.

Each period node in the graph gets a randomly chosen part of the total computational time assigned to it. Distributing the value may be done in three different ways. Firstly a uniform distribution, where the total computational time is divided by the number of periods. Secondly a pure random distribution, where the total computational time is randomly distributed to the periods. Lastly a manual configured distribution with an array of percentages, where the total computational time is distributed to the periods according to the given percentages.

When distributing the computational time, the user shall keep in mind that the value per period is getting split among the repetitions during the planning cycle. Meaning with an example of a period of 1000ms and a computational time of 500ms, the computational time is split among the repetitions of the period. In a case of a hyper-period of 16000ms, the period of 1000ms is repeated 16 times, and the computational time of 500ms is split among the repetitions. Resulting in a computational time of 31.25ms per repetition of the period, in which case the computational time is rounded down to 31ms. This may be a slight differ from the expected value that should be kept in mind, when checking for the utilization of the system.

Another point to consider is the possible lack of of variation if each layer gets very small values assigned. Since per period multiple tasks may be generated, the assigned value shall be high enough to encompass these tasks.

The result of this operation is a graph with the periods and the computational time assigned to each period.

4.6 Spanning the tree

Having until now a tree with the periods and the computational time assigned to each period, the next step is to span the tree. This is done by calculating the amount of repetitions of each period and inserting a new node per repetition into the graph. Due to each node representing a time slot of the planning cycle, each repetition gets assigned an index to be able to keep them in order.

To connect the nodes into a tree, all nodes per period are compared to all nodes of the next higher period. Knowing the factor between these periods it is a simple modulo calculation to assign the nodes in order to mirror their association between each other. Clearly visible in Section 3.2.1 the higher period gets as many children of the lower period as the factor between the periods is.

Having the graph spanned, the result is a tree with the time slots of each period represented by a node and its subtree, yet without any assigned computational time to any instance of a period. Due to the many layers to far for randomly assigning values and amounts, it is a difficile challenge to take the computational time assigned to the periods and distribute it among the repetitions. Varying the amount per repetition burdens many checks to ensure the represented part of the planning cycle is still schedulable. Some checks are simple, but challenging in combination, here is an example of two conditions that must be met.

1. Every period instance shall be assigned a computational value above or equal to 1.
2. Every period instance shall be assigned a computational value lower or equal to the period itself.
3. The assigned computational values of all children of a parent node shall not equal or exceed the period of the parent node.

The first and second condition are simple checks to ensure that the computational time assigned to a period is schedulable within the period itself and contain any time to be assigned to the later generated tasks.

The third condition is a bit more complex, but still simple to understand. It limits the computational amount the children of a node may have, to ensure that the parent node is schedulable. If no such check would be in place, the parent node would be assigned a computational time that would not fit into the period itself, hence making the parent node unschedulable. As example consider a small tree, consisting out of seven nodes:

- A root node with a period of 4000ms.

- Two children nodes with a period of 2000ms.
- Four leaf nodes with a period of 1000ms.

With a hyperperiod of 4000ms and an utilization of 1, the computational time to distributed equals 4000ms. Assume a distribution of 250ms for the top-most layer, 750ms for the second layer, 3000ms for the last layer. In the last layer a problematic distribution would be one, in which all period instances with the same parent node would be assigned a computational time of 1000ms. If the first two nodes in the last layer get assigned 1000ms, the first node in the second layer would not have any time left to get this scheduled in any way. This would result in a non-schedulable system, which is not the goal of the generation process.

While this is a simple and comprehensible problem with easy solutions in this small example, it gets more complex with more layers and nodes, since each added layer would at least add double the amount of leaf nodes. For this reason the distribution among instances of the same period is done in this implementation by a simple uniform distribution.

The result is a tree representing the planning cycle and the time slots in which the in Section 4.7 generated tasks, respectively jobs, have to be scheduled, including their assigned WCET.

4.7 Tasks and Chains

The next step is to generate chains of tasks, that communicate between each other and their respective task-set including the jobs to be scheduled, based on the idea presented in Section 3.2.2.

The process of generating tasks and chains begins by iterating over all represented periods in the tree. For each period, a new and empty task is added to the first chain. This ensures that every period has a corresponding task in the initial chain. Subsequent chains are created with a random skip variable, which is a configuration parameter that determines whether a period should be skipped. This introduces variability in the task generation process.

The random skip variable is configured to control the likelihood of skipping a period. This can be adjusted to achieve the desired level of randomness in the task chains. By varying the skip parameter, different task sets can be generated, each with unique characteristics. This flexibility allows for the creation of diverse benchmark programs that can be used to evaluate the performance of scheduling algorithms.

For now the generated tasks and chains are mere placeholders without being usable for anything else. To correct this, each task, associated with its period, is assigned a random WCET value. This is done by taking the in Section 4.6 assigned computational time per period and splitting it randomly among its assigned tasks.

For now each task has assigned a WCET value, the task-set is per definition schedulable while keeping the utilization of the system at 1, or the desired value. The task-set is schedulable, since all previous steps assured to keep the computational time at a maximum within its limits defined by the range of the period. More on scheduling in the next **??**.

For now this task-set is the same result as if each tasks WCET has been estimated by a WCET analysis tool. But to create a model, that can be used to test and validate real-time scheduling algorithms, the tasks need to be assigned with a variation of execution times based on the idea presented in Section 3.2.5. Since it is known for the WCET of the task to be schedulable, every lower execution time has to be schedulable as well. For that reason each tasks WCET is split into a range of values and assigned as a possible execution time that a job of the task may have.

To implement the concept of conditional runtimes presented in Section 3.2.5, each tasks and its successors of the corresponding chain list of possible execution times are laid next to each other, with one list shuffled, and iterated over to create random combinations of possible predecessor and successor. If one of the lists is shorter than the other, the shorter list is repeated until the longer list is exhausted. The intention was to repeat this for certain configurable amount to create a known multiple of possible follow ups and chain variations.

Sadly the current implementation does not yet support the use of that conditions in the schedule, but it is planned to be implemented in the future. Furthermore is the use of task independent input values planned to simulate the use of sensors and other external influences on the tasks execution time.

4.8 Results and Output

simple image depicting the filling?

Like described in the concept of flattening the tree into a schedule in Section 3.2.8 the tree is traversed depth-first while creating schedule entries for each node. For each leaf node, representing the highest period a separate schedule container is created. This container is filled with the jobs of the leaf node, which are then scheduled in the time slots of the planning cycle. After every leaf node is inserted in such schedule, the schedule is passed to the parent node, which then inserts its own jobs into the child schedules in order of the time slots. If a child schedule still has idle time slots, the jobs of the parents node are inserted into the child schedule until all time slots are occupied or the parent node has no jobs left to insert. After one child schedule is filled, the next child's schedule is appended to the previous schedule, repeating the insertion of jobs until all child schedules are processed or the parent node has no jobs left to insert. The result is a combined schedule of the child nodes and the parent node, which is then passed to the parent node, repeating the process until the root node is reached.

The result is a list representing the planning cycle with the jobs scheduled, exported as a CSV for further use.

5 Summary and Discussion

5.1 Summary

The motivation for this thesis was to address the limitation of existing benchmarks. These benchmarks often do lack a variation of complexity, e.g. inter-task communication and

5.2 Discussion

In the introduction four research questions were presented, which are addressed in the following Sections 5.2.1 to 5.2.4.

5.2.1 Reflection of real-world systems

5.2.2 Flexibility

5.2.3 Correctness

5.2.4 Improvement of Accuracy

5.3 Future Work

The current implementation has some limitations and unimplemented planned features to be areas for future work.

5.3.1 Harmonic Task Sets

The current implementation of the task set generator is limited to harmonic task sets. To expand the generator to non-harmonic task sets, the generator must be extended to support the generation of arbitrary task sets. This extension would require a more complex algorithm, since the overlapping of timing windows complicate the planning by a great margin. The benefit of using a tree would no longer be applicable, since the tree would not be able to represent the overlapping of timing windows. Maybe by splitting the overlap and mapping them with the corresponding period frames, a modified tree could be used to represent the task set.

5.3.2 Conditional Chains

Sadly the current implementation does not yet support the use of that conditions in the schedule, but it is planned to be implemented in the future. Furthermore is the use of task independent input values planned to simulate the use of sensors and other external influences on the tasks execution time.

5.3.3 Export to AMALTHEA

Add more dependencies and used resources. The in Section 4.6 presented distribution of computational time among instances of the same period still is a simple uniform distribution. Whilst working fine, there may be a need for a more complex distribution depending on future requirements.

Export to AMALTHEA Expand to non harmonic task sets.

A Acronyms

AI	Abstract Interpretation	10
BCET	best-case execution time	11
BET	Bounded Execution Time	7
CFG	control-flow graph	9
DFA	data flow analysis	9
IPET	implicit-path enumeration	9
LCM	least common multiple	15
LET	Logical Execution Time	7
LRU	Least Recently Used	9
RNG	random number generator	23
RMS	Rate Monotonic Scheduling	12
RTOS	Real-Time Operating System	5
WCET	worst-case execution time	iii

WCRT	worst-case response time	2
ZET	Zero Execution Time	7

B Notation

Name	Description
τ_i	Task i
$\tau_{i,k}$	k-th Job of task i
$\Gamma = \{\tau_1, \tau_2, \dots, \tau_n\}$	Task-set with n tasks
$\Gamma^c = \{\tau_1, \tau_2, \dots, \tau_n\}$	Chain c with n tasks
$\Gamma^c[i]$	i-th task of chain c
C_i	WCET of task i
T_i	Period of periodic task i
$r_{i,k}$	Release time of $\tau_{i,k}$

Table B.1: Used Notations

Bibliography

[1] Jakaria Abdullah, Gaoyang Dai, and Wang Yi. "Worst-Case Cause-Effect Reaction Latency in Systems with Non-Blocking Communication." In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1625–1630. ISBN: 3-9819263-2-3.

[2] L. Abeni and G. Buttazzo. "Integrating Multimedia Applications in Hard Real-Time Systems." In: *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*. Madrid, Spain: IEEE Comput. Soc, 1998, pp. 4–13. ISBN: 978-0-8186-9212-3. DOI: `10.1109/REAL.1998.739726`. (Visited on 02/13/2025).

[3] Matthias Becker et al. "Synthesizing Job-Level Dependencies for Automotive Multi-Rate Effect Chains." In: *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2016, pp. 159–169. ISBN: 1-5090-2479-4.

[4] Giorgio Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Cham: Springer Nature Switzerland, 2024. ISBN: 978-3-031-45409-7 978-3-031-45410-3. DOI: `10.1007/978-3-031-45410-3`. (Visited on 02/22/2025).

[5] Samarjit Chakraborty and Jörg Eberspächer, eds. *Advances in Real-Time Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-24348-6 978-3-642-24349-3. DOI: `10.1007/978-3-642-24349-3`. (Visited on 02/26/2025).

[6] Hyunjong Choi, Mohsen Karimi, and Hyoseung Kim. "Chain-Based Fixed-Priority Scheduling of Loosely-Dependent Tasks." In: *2020 IEEE 38th International Conference on Computer Design (ICCD)*. Hartford, CT, USA: IEEE, Oct. 2020, pp. 631–639. ISBN: 978-1-7281-9710-4. DOI: `10.1109/ICCD50377.2020.00109`. (Visited on 02/26/2025).

[7] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints." In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages - POPL '77*. Los Angeles, California: ACM Press, 1977, pp. 238–252. DOI: `10.1145/512950.512973`. (Visited on 02/15/2025).

[8] Dar-Tzen Peng, K.G. Shin, and T.F. Abdelzaher. "Assignment and Scheduling Communicating Periodic Tasks in Distributed Real-Time Systems." In: *IEEE Transactions on Software Engineering* 23.12 (Dec. 1997), pp. 745–758. ISSN: 00985589. DOI: `10.1109/32.637388`. (Visited on 11/07/2024).

[9] Z. Deng and J.W.-S. Liu. "Scheduling Real-Time Applications in an Open Environment." In: *Proceedings Real-Time Systems Symposium*. San Francisco, CA, USA: IEEE Comput. Soc, 1997, pp. 308–319. ISBN: 978-0-8186-8268-1. DOI: `10.1109/REAL.1997.641292`. (Visited on 02/13/2025).

[10] Christian Dietrich et al. "SysWCET: Whole-System Response-Time Analysis for Fixed-Priority Real-Time Systems." In: ().

[11] Christian Eichler, Peter Wägemann, and Wolfgang Schröder-Preikschat. "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*. Montreal Quebec Canada: ACM, Apr. 2019, pp. 1–6. ISBN: 978-1-4503-6693-9. DOI: `10.1145/3312480.3313170`. (Visited on 11/07/2024).

[12] Christian Eichler et al. "TASKers: A Whole-System Generator for Benchmarking Real-Time-System Analyses." In: *OASIcs, Volume 63, WCET 2018* 63 (2018), 6:1–6:12. ISSN: 2190-6807. DOI: `10.4230/OASICS.WCET.2018.6`. (Visited on 11/07/2024).

[13] Heiko Falk et al. "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research." In: *OASIcs, Volume 55, WCET 2016* 55 (2016), 2:1–2:10. ISSN: 2190-6807. DOI: `10.4230/OASICS.WCET.2016.2`. (Visited on 11/07/2024).

[14] Jan Gustafsson et al. "The Mälardalen WCET Benchmarks: Past, Present And Future." In: *OASIcs, Volume 15, WCET 2010* 15 (2012), pp. 136–146. ISSN: 2190-6807. DOI: `10.4230/OASICS.WCET.2010.136`. (Visited on 02/22/2025).

[15] Prasanna Hambarde, Rachit Varma, and Shivani Jha. "The Survey of Real Time Operating System: RTOS." In: *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*. Nagpur, India: IEEE, Jan. 2014, pp. 34–39. ISBN: 978-1-4799-2102-7. DOI: `10.1109/ICESC.2014.15`. (Visited on 02/13/2025).

[16] Tarek Helmy. "Optimizing Round-Robin Scheduling Algorithm Performance in Real-Time Systems." In: *Applied Intelligence and Informatics*. Ed. by Mufti Mahmud et al. Vol. 2065. Cham: Springer Nature Switzerland, 2024, pp. 371–382. ISBN: 978-3-031-68638-2 978-3-031-68639-9. DOI: `10.1007/978-3-031-68639-9_24`. (Visited on 02/10/2025).

[17] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. "Giotto: A Time-Triggered Language for Embedded Programming." In: *Proceedings of the IEEE* 91.1 (Jan. 2003), pp. 84–99. ISSN: 0018-9219. DOI: `10.1109/JPROC.2002.805825`. (Visited on 02/28/2025).

[18] *IEEE Standard for a Real-Time Operating System (RTOS) for Small-Scale Embedded Systems*. DOI: `10.1109/IEEESTD.2018.8445674`. (Visited on 02/13/2025).

[19] Timon Kelter. "WCET Analysis and Optimization for Multi-Core Real-Time Systems." In: ().

[20] T.-W. Kuo and A.K. Mok. "Load Adjustment in Adaptive Real-Time Systems." In: *[1991] Proceedings Twelfth Real-Time Systems Symposium*. San Antonio, TX, USA: IEEE Comput. Soc. Press, 1991, pp. 160–170. ISBN: 978-0-8186-2450-6. DOI: `10.1109/REAL.1991.160369`. (Visited on 02/27/2025).

[21] J. Lehoczky, L. Sha, and Y. Ding. "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior." In: *[1989] Proceedings. Real-Time Systems Symposium*. Santa Monica, CA, USA: IEEE Comput. Soc. Press, 1989, pp. 166–171. ISBN: 978-0-8186-2004-1. DOI: `10.1109/REAL.1989.63567`. (Visited on 01/19/2025).

[22] John P. Lehoczky and Lui Sha. "Performance of Real-Time Bus Scheduling Algorithms." In: *Proceedings of the 1986 ACM SIGMETRICS Joint International Conference on Computer Performance Modelling, Measurement and Evaluation - SIGMETRICS '86/PERFORMANCE '86*. Raleigh, North Carolina, United States: ACM Press, 1986, pp. 44–53. ISBN: 978-0-89791-184-9. DOI: `10.1145/317499.317538`. (Visited on 02/03/2025).

[23] Y.-T. S. Li, S. Malik, and A. Wolfe. "Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches." In: *17th IEEE Real-Time Systems Symposium*. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1996, pp. 254–263. ISBN: 978-0-8186-7689-5. DOI: `10.1109/REAL.1996.563722`. (Visited on 02/23/2025).

[24] Chung Laung Liu and James W Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment." In: *Journal of the ACM (JACM)* 20.1 (1973), pp. 46–61. ISSN: 0004-5411.

[25] Vijayshree Shinde and Seema C. "Comparison of Real Time Task Scheduling Algorithms." In: *International Journal of Computer Applications* 158.6 (Jan. 2017), pp. 37–41. ISSN: 09758887. DOI: `10.5120/ijca2017912832`. (Visited on 02/03/2025).

[26] John A Stankovic and Raj Rajkumar. "Real-Time Operating Systems." In: *Real-Time Systems* 28.2-3 (2004), pp. 237–253. ISSN: 0922-6443.

[27] Reinhard Wilhelm et al. “The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools.” In: *ACM Transactions on Embedded Computing Systems* 7.3 (Apr. 2008), pp. 1–53. ISSN: 1539-9087, 1558-3465. DOI: `10.1145/1347375.1347389`. (Visited on 02/14/2025).

Eidesstattliche Versicherung

(Affidavit)

Brockmann, Henning	141194
Name, Vorname (surname, first name)	Matrikelnummer (student ID number)

Bachelorarbeit (Bachelor's thesis) [✓ Masterarbeit (Master's thesis)]

Titel (Title)

Generating Real-Time System Specifications with Known Response Times

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem oben genannten Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present thesis with the above-mentioned title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution before.

Dortmund, 03.03.2025	[illegible]
Ort, Datum (place, date)	Unterschrift (signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to EUR 50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the Chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, Section 63 (5) North Rhine-Westphalia Higher Education Act (*Hochschulgesetz, HG*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund University will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:*

Dortmund, 03.03.2025	[illegible]
Ort, Datum (place, date)	Unterschrift (signature)

*Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.