Delft University of Technology
Master of Science Thesis in Embedded Systems

Investigating Non-Work-Conserving Scheduling in Event-Driven Real-Time Systems

Nathan Jacobus van Ofwegen
Supervisor: Dr. Mitra Nasri

TU Delft Delft University of Technology

Investigating Non-Work-Conserving Scheduling in Event-Driven Real-Time Systems

Master of Science Thesis in Embedded Systems

Embedded and Networked Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Nathan Jacobus van Ofwegen

26-10-2020

Author
Nathan Jacobus van Ofwegen

Title
Investigating Non-Work-Conserving Scheduling in Event-Driven Real-Time Systems

MSc Presentation Date

02-11-2020

Supervisor
Dr. Mitra Nasri

Graduation Committee

Dr. Mitra Nasri	Eindhoven University of Technology
Dr. Ir. Marco Zuiniga	Delft University of Technology
Prof. Dr. Ir. Alexander Verbraeck	Delft University of Technology

Abstract

Embedded real-time systems that have cost or energy constraints are usually limited in processing power and memory. This limitation typically leads to applying simpler execution models such as non-preemptive scheduling. A problem with a non-preemptive real-time system is that finding a schedule without causing deadline misses is NP-hard. Finding a schedule is therefore done with online scheduling policies, i.e. policies that make decisions upon arrival or after execution of a task to schedule the next one.

Online non-preemptive scheduling policies are typically priority based, namely, they pick the highest priority job to schedule based on criteria as period or absolute deadline. These are work-conserving policies which cannot keep the processor idle while there are still pending jobs in the ready queue.

Non-work-conserving policies allow an idle cpu while there are still jobs in the ready queue. While this increases schedulability, it also has an increased overhead. Current state of the art policies like Precautious Rate Monotonic (PRM) and Critical Windows Earliest Deadline First (CW-EDF) have an idle-time insertion policy which can insert an idle time in the schedule (between the execution of the jobs) while still having pending jobs. PRM verifies whether the highest priority job in the ready queue is able to finish without causing a deadline miss for the highest priority task in the system, which is the task with the lowest period. PRM improves schedulability with increased overhead of $O(1)$. CW-EDF comes with additional overhead $O(n \log n)$ in which it verifies if scheduling the highest priority pending job will result in a deadline miss for the next job of all other tasks in the system. PRM has low overhead, while CW-EDF has better schedulability despite having higher overhead.

A limitation of these non-work-conserving policies is the missing support for event-triggered task, where jobs are released at unknown time instants. Namely, the existing non-work-conserving policies are designed for strict periodic tasks.

In this thesis we will introduce a policy which has schedulability as high as CW-EDF, but prior to running the system it detects the critical tasks which have an influence on the idle-time insertion policy. We ignore the non-critical tasks during the idle-time insertion policy, reducing the runtime overhead. We also introduce a policy which will support time-triggered tasks, by using an arrival curve which stores the possible behavior of the event-triggered task. With this arrival curve we will reduce the number of deadline misses compared to the existing non-work-conserving policies.

Preface

Sisu alkaa sieltä, missä sinnikkyys loppuu

I want to thank Mitra Nasri for supervising and the rest of the RTS and ENS group for providing feedback during this time, I learned a lot. Massive thanks to The Barn and TU Delft Gamelab for providing me with their workplaces, resources and friendships. More specifically, thanking Arne Bezuijen for his feedback during the last weeks of writing this thesis and listening to whatever I had to say during the corona walks. Thanking my parents for their love during the entire masters program. I'm forever grateful for their involvement during the fall and rise in difficult times. Finally, I want to thank my bunny Midi for giving me the best cuddles and the ability to unwind, *mlem*.

Nathan Jacobus van Ofwegen

Delft, The Netherlands
26th October 2020

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded systems that have cost or energy constraints usually have limited processing power and memory. Because of the processing and memory limitations, these systems often have no operating system and don't support multitasking. A downside is that processes that run on these systems cannot be halted or stopped in order to run other processes. Allowing switching between processes costs memory, time and processing power. As a result, these embedded systems are likely to run in a non-preemptively environment, meaning that running processes cannot be halted or paused.

A problem in non-preemptive systems is that we need to be sure that we pick the right order of executing processes (tasks). If there are multiple tasks in the ready queue, the scheduling must pick the correct task, while making sure other tasks will still finish execution before their deadline. To make these decisions, the system requires knowledge of the future. A solution to find this order is to brute force all the possibilities of the tasks in the system to find a schedule, but this solution is exponential in relation to the number of tasks in the system. Jeffay et al. [7] has shown that finding such a schedule for non-preemptive periodic tasks on uniprocessor platforms is NP-Hard.

Non-Preemptive Earliest Deadline First (NP-EDF) and Non-Preemptive Rate Monotonic (NP-RM) [5] are online work-conserving scheduling algorithms for non-preemptive systems. Work-conserving algorithms never leave the processor idle while there are tasks available to be executed. An issue with work-conserving algorithms is that if a high-priority task is released just after a lower-priority has started on the processor, the high-priority must wait for the lower-priority task to complete. This leads to *priority inversion*, which can easily cause the high-priority task to miss its deadline.

To solve this problem of work-conserving non-preemptive policies, Nasri et al. [11] introduced a policy called Precautious Rate Monotonic (PRM). A key feature is that PRM introduces an Idle-time Insertion Policy (IIP) on top of NP-RM. The IIP decides whether to execute the highest-priority task in the ready queue or to leave the processor idle (non-work-conserving). In order to take this decision, the IIP looks into the future and checks whether executing the highest-priority task will cause a deadline miss *for the next instance of the task with the*

smallest period in the system. In terms of overhead, PRM requires to check only one extra condition for the scheduling decision, adding only a constant overhead compared to NP-RM. However, this policy only works for strictly periodic tasks.

Another non-work-conserving policy is Critical Window Earliest Deadline First (CW-EDF) [10]. This policy is comparable with PRM, but its IIP requires more calculations. While this improves the schedulability compared to PRM, it comes with polynomial overhead $O(n \log n)$ with respect to the number of tasks in the system. Both policies are explained in detail in section 3.3. This thesis will use these two non-work-conserving policies as a baseline.

These two non-work-conserving policies focus both on different metrics. PRM focuses on keeping the overhead low at $O(1)$, while still achieving better schedulability than work-conserving non-preemptive policies. CW-EDF focuses on improving the schedulability, but this comes with an polynomial overhead $O(n \log n)$. Currently there is no policy that fits in between, aiming for improved schedulability while maintaining a low overhead.

Another main limitation of the state of the art on non-work-conserving policies is the support of event-driven systems. Event-driven systems have tasks (event-triggered) that arrive dynamically at unknown time instants. This leads to an unknown future, which current policies require for their scheduling decisions. PRM and CW-EDF are designed for strictly periodic tasks and cannot work for event-triggered tasks.

Finding an algorithm which adapts to a dynamic release pattern is valuable as Akesson et al. [1] showed that over 60% of real-time systems have event-triggered task activation.

In this thesis we try to improve two main aspects of the problem described above. Namely the reduction of runtime overhead and adding support for event-driven tasks. We aim to answer the following research questions (**RQ**):

- **RQ 1**: *How can we lower the run-time overhead of CW-EDF but maintain its schedulability?*
- **RQ 2**: *How can we add support for event-triggered tasks in a non-work-conserving policy?*

For **RQ 1**, we will use the schedule generated by CW-EDF but we will reduce the overhead by identifying and skipping non-critical tasks which are considered in the scheduling decision in CW-EDF. Non-critical tasks are tasks that have no impact on the decision of the idle-time insertion policy.
For **RQ 2**, we will use a small table which stores the possible behavior of the event-triggered task. For the scheduling decisions, we will use this table to find a realistic situation the system might face in the future. For systems where a deadline miss is not critical, we will try to reduce the number of deadline misses using this table, increasing the value of the system.

The remainder of this thesis is structured as follows: We start with a short background about real-time systems which covers the basic concepts for the

rest of this thesis in Chapter 2. Chapter 3 will describe the current state-of-the-art solutions for our problems and explain the shortcomings in detail. The system model will be presented in Chapter 4, explaining the foundation and exact environment of our work. In Chapter 5 and 6 we present our solution for the problems described above. Next, we evaluate the given solutions in Chapter 7 and finally the conclusion and future work will be presented in Chapter 8.

Chapter 2

Background

In this chapter we provide background information in order to understand the basics of real-time systems. We start with the basic concepts of real-time systems with their tasks and followed by how these tasks are handled by the scheduling policies.

2.1 Tasks

Each real-time system contains a set of *tasks*. A task can create an unlimited number of instances, called a *job*, which are executed on the CPU. The time a task becomes ready for execution we call the *arrival time* or *release time*. A task that is currently running on the CPU is called an active task. Tasks that are waiting for the CPU to become available are *pending* and are kept in a ready queue. If the system activates a task from the ready queue we *dispatch* the task. Each job has a *deadline* by which it must complete its execution in order to guarantee correctness of the system. Tasks have different activation patterns, which describe the way tasks are releasing their jobs. Also their constraints and execution method can differ.

2.1.1 Activation patterns

Tasks are released in different patterns. They can be released with a fixed or dynamic interval.

Periodic tasks Periodic tasks are tasks that arrive regularly at a constant rate. The interval between two consecutive arrivals is the period of the task. We also call these tasks time-triggered.
As an example, reading out sensors is often a periodic task. For instance the sensors in cars which detect a collision or altitude sensors in avionics. These sensors are typically processed every couple of milliseconds [8].

Sporadic tasks Sporadic tasks are tasks with irregular inter-arrival time, but have a minimum inter-arrival time between two jobs.

Aperiodic tasks Aperiodic tasks are tasks that consist of identical jobs, but their activation times are not regularly interleaved [2]. We also call these tasks event-triggered. As an example, handling user input is an event-triggered task. Mouse clicks are rarely pressed in a periodic fashion, the system never knows exactly when the user will click or press a button. Also alarms are event-triggered, the arrival of handling an emergency task are obviously unknown to a system but have to be handled correctly. We can model the activation pattern of an event-triggered task with an arrival curve, which plays an important role in our work.

Arrival curve An arrival curve represents the lower and upper bound on the inter-arrival time between consecutive events of the event-triggered task. We use the arrival curve to store the measured behavior of a task. Each consecutive number of events has its own lower and upper bound.

As an example, we measured the release events of a task at the following timestamps: 2, 10, 13, 15, 20, 24. For the first lower and upper bound between *two* consecutive events, we want to know the maximum and minimum interval ever occurred during the measurements. Looking at Figure 2.1, we see that the lowest interval is 2 (between 13 and 15). The same for the maximum, which is 8 (between 2 and 10). We store these two values on the curve at point a and b in Figure 2.2. We do the same for *three* consecutive events. We see that the minimum interval between three consecutive events is 5 (between 10 and 15) and the maximum interval is 11 (between 2 and 13). We store these values again, at point c and d respectively. We continue this process until we have created the entire arrival curve. Since we only have measured 6 release events, the arrival curve cannot go higher than 6 consecutive events.

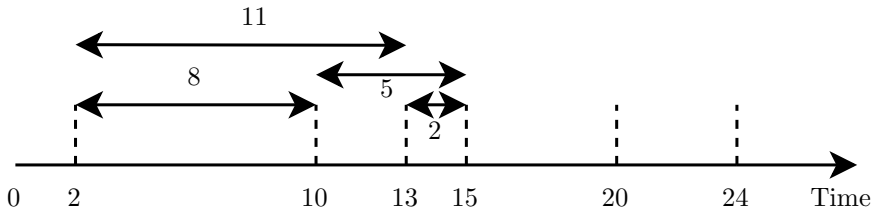


Figure 2.1: Possible intervals between consecutive events

By calculating the arrival curve we have information regarding the variable period of the event-triggered task, over different interval lengths. With this information we are now able to guarantee that the next arrival won't be *before* the lower bound and that the next arrival won't be *after* the upper bound.

In order to compute the lower and upper bound we need a lot of samples of the arrivals of the event-triggered task. The more samples there are, the more accurate the arrival curve. Measuring the arrival events of these event-triggered tasks are widely used in the industry. Because these measurements are performed before execution of the system they are not counted as overhead. Recently Carvajal et al. [3] have introduced a generation algorithm which allows the generation of arrival curves with a event size up to 5 orders of magnitude

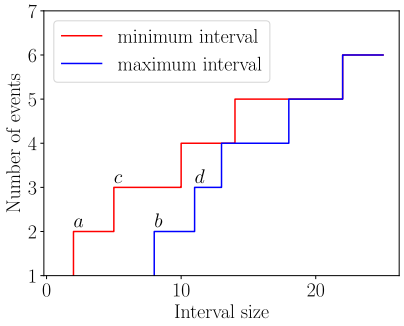


Figure 2.2: Arrival curve of an event-triggered task

in a relatively short amount of time (few minutes). While this is very fast, the generated arrival curve is typically too big to fit into the small memory of embedded systems. The table and history size needed will be analyzed in Chapter 6. Generating the curve is done using a self created algorithm and is explained in section 6.1.

2.1.2 Timing constraints

The typical constraint we mentioned is the deadline, a timing constraint, which is the time in which the task must complete its execution without causing any negative effects on the system. Missing the deadline has different consequences in different type of systems.

Soft In soft real-time systems the deadlines are not strict deadlines. If a task fails to finish before the deadline it is not considered a failure, but the value of the system drops. The time the task finished after the deadline is called tardiness. In a soft real-time system we will try to minimize this tardiness, but not prevent it.
An example of a soft real-time system is a video conference call. If the video fails for a brief moment, the user will experience some stutter in the system. While we don't want the system to stutter, it's not considered a failure if we lose a few frames.

Hard In hard real-time systems the deadlines are strict. If a task misses its deadline, the system will be considered a failure because it might cause catastrophic consequences.
The airbag system of a car is often used as an example for a hard real-time task. We want the airbag to be deployed at exactly the right time. If the airbag deploys either too soon or too late, it can have catastrophic consequences to the driver.

2.1.3 Execution model

Two relevant execution models for our work are preemptive and non-preemptive.

Preemptive execution Preemptive tasks are tasks that can go from the running state back to the ready queue, many operating systems allow this. If an high-priority task arrives in the ready queue it can immediately be executed since the lower-priority running task can be put back into the ready queue.

Non-preemptive execution Non-preemptive tasks don't allow preemption. Running tasks cannot be suspended by other higher-priority tasks.

2.1.4 Schedule

To visualize the task execution order of our system, we will use a schedule. In each figure the x-axis will represent the time and the y-axis will represent the different tasks in a system. We will use up-arrows for arrivals and down-arrows for deadlines of the tasks. If the deadline is at the same time as the arrival of the next job, only up-arrows are used. The boxes represent the dispatching and execution of a task on the processor.
In Figure 2.3 we will have a simple system with two periodic tasks. Task 1 has a period and deadline of 4 and computational time of 2. Task 2 has a period of 6, a deadline of 5 and a computational time of 3. In this example we are on an uniprocessor, meaning that two tasks cannot execute simultaneously so they don't overlap in the schedule.

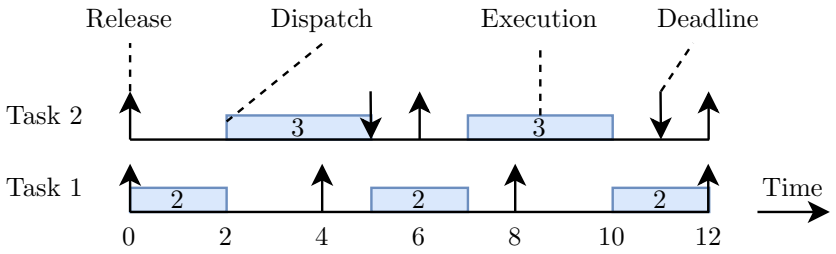


Figure 2.3: Visualization of a schedule

2.2 Scheduling policies

While the example in Figure 2.3 is simple, real life situations have more complex task sets and the system needs to carefully select which task will be dispatched. A scheduling policy will decide which job is dispatched from the ready queue.

2.2.1 Classification

There are a great variety of scheduling algorithms, we consider the following main classes of policies:

Non-preemptive vs Preemptive In non-preemptive systems, an active task is executed by the processor until completion. Because of this, all scheduling decisions by the policy are taking after the execution of a task and at the arrival of a task. However, the policy cannot make any decision on arrival if the

processor already has an active task running. Preemptive policies are able to switch between tasks, while they are still busy executing. On preemption the current running task is put into the memory and the system will run the other task. After finishing that task, the system can return to the previous task and continue its execution.
Preemptive systems require more memory and switching between these tasks takes additional time. Embedded systems typically don't have the resources to deal with preemption. These systems are non-preemptive systems and use non-preemptive policies.

Offline vs Online Offline policies are executed on the entire task set before execution. This results in a schedule which is stored in a table and during actual execution of the system, the table is accessed to dispatch the next task.
Online policies are policies where scheduling decisions are taken at runtime on the arrival of a task or when an active task finishes execution. For preemptive tasks this decision is taken on both the release and finishing of an execution of a task. For non-preemptive tasks it is only when the active tasks finishes execution or on arrival when the system has no active task executing.

Non-work-conserving A non-work-conserving policy is a policy that may decide to leave the processor idle even though there might be pending tasks in the ready queue. While it seems counterintuitive to have an idle system while there is work to do, it has its advantages in avoiding blocking which otherwise could cause a deadline miss for the task that will soon be released. Policies that have these advantages will be explained in Chapter 3.

2.2.2 Performance metrics

There are important metrics that we will use in this thesis to measure the performance of a policy to make a decision. These metrics indicate if a policy can find a schedule for a given task set and how much computational resources it took to find a schedule.

Schedulability A task set is schedulable by policy A if the policy A guarantees that for any admissible workload pattern that can be generated from that task set, the schedule generated by the policy will never violate their task constraints.

Runtime overhead The overhead is the time spent by scheduling policy. Overhead is to be minimized, since it uses computational resources which otherwise can be used by the tasks themselves.

Deadline misses For soft real-time systems, a deadline miss is not critical but the system loses some of its value. Therefore the number deadlines misses are to be minimized by a policy, maximizing the value of a system.

Chapter 3

Related Work

In this chapter we explain different scheduling policies which are related and form the basic of our work. The shortcomings will be mentioned, with possible solutions, to clarify the need of this thesis. We start with offline scheduling techniques in Section 3.1. Next we show online policies, where we start with work-conserving policies in section 3.2. After that we show two non-work-conserving policies which introduces idle-time insertion in section 3.3. The first one is Precautious Rate Monotonic (PRM) and the second one is Critical Window Earliest Deadline First (CW-EDF). PRM focuses on a low overhead, while CW-EDF focuses on higher schedulability.

3.1 Offline scheduling generation techniques

There are scheduling solutions which synthesize the schedule before run-time. These solutions can guarantee in advance that the deadline will be met. Possible solutions are brute force algorithms, using a search tree [14] or integer linear programming (ILP) [13]. All these solutions try to solve an NP-hard problem and result in a schedule we need to store the schedule on the memory of the embedded system. Memory is a resource limitation for embedded systems and typically don't have the resources to store this generated schedule. Hence these solutions are not the focus of this thesis.

3.2 Work-conserving policies

In this section we explain two similar non-work-conserving policies, namely Non-Preemptive Rate Monotonic (NP-RM) and Non-Preemptive Earliest Deadline First (NP-EDF). Both have a very low overhead of $O(1)$ and $O(n)$ respectively.

Policy NP-RM and NP-EDF are non-preemptive scheduling policies with a low overhead. NP-RM dispatches the next pending job based on the period of the task. A shorter period will result in a higher priority. The highest priority task is determined before run-time, hence the overhead of $O(1)$. Non-Preemptive Earliest Deadline First (NP-EDF) will dispatch the next task based on the earliest deadline of the pending jobs. Since NP-EDF will have to determine the

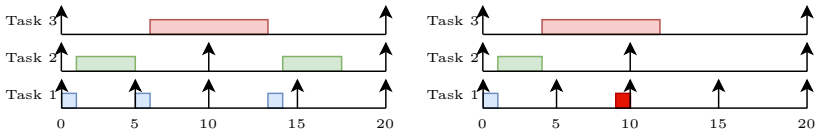
Task	Priority	Period	Computational time	Relative deadline
1	1	5	1	5
2	2	10	4	10
3	3	20	8	20

Table 3.1: Tasks used in Figure 3.1a

current job with the earliest deadline during run-time, it has an overhead of $O(n)$ with respect to the number of tasks.

Example An example of a NP-RM schedule is shown in Figure 3.1a. This system contains three tasks described in Table 3.1. These tasks have an relative deadline equal to the period.

At the start all tasks are released and pending. Since task 1 has the lowest period of the three tasks, it has the highest priority and it is dispatched by the scheduling policy. After the execution of task 1, task 2 and 3 are the only tasks pending and the scheduling policy dispatches task 2 because of the lowest period and thus highest priority. After the execution of task 2, task 1 is again pending. The system now has two pending jobs, namely task 1 and task 3. Since task 1 has a higher priority it gets dispatched instead of task 3. After execution task 1 we see that task 3 is the only pending task in the system, so task 3 has the highest priority and gets dispatched.
In short, NP-RM dispatches the pending task with the shortest period, keeping the overhead low.



(a) NP-RM schedule (b) NP-RM anomaly

Figure 3.1: Schedules using NP-RM

Shortcomings If we change the task set where the execution of task 2 is shorter, we will encounter a deadline miss for task 1, as seen in figure 3.1b. We see that after the execution of task 2 (at time 4), task 3 gets dispatched because it is the only pending task in the system at that point. But execution of task 3 results in a deadline miss of the next job of task 1. There is however a schedule that misses no deadlines for this task set by waiting a short time for task 1, as presented in the next section.

3.3 Non-work-conserving policies

In this section we will explain two state of the art non-work-conserving policies, namely Precautious Rate Monotonic and Critical Window Earliest Deadline

First. In these policies we will check if we can schedule the current task without causing any deadline misses. With PRM we will check if we don't cause a deadline miss for the highest priority task in the system. For CW-EDF we will do this for all other tasks that are not in the ready queue.

3.3.1 Precautious Rate Monotonic

PRM [11] is a non-work-conserving policy which inserts idle time in between jobs, while there are still pending tasks. In contrast to NP-RM, where the scheduler will dispatch a pending task whenever the CPU is available, PRM uses an Idle-time Insertion Policy (IIP). This IIP verifies if the current highest priority pending job can finish execution without causing a deadline miss for the next job of the highest priority task, which is the task with the lowest period in the system. Due to the constant overhead added, PRM has an overhead of $O(1)$.

Notations This section will introduce some notations which are used, illustrated in Figure 3.2. The complete set of notations used in thesis are located on page 55.

Symbol	Description
t	Current time
τ_i	Task i
T_i	Period of task τ_i
C_i	Computational time of task τ_i
D_i	Deadline of task τ_i

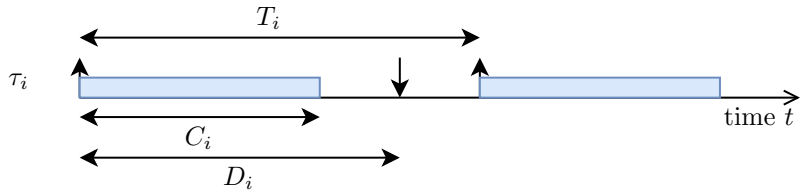


Figure 3.2: Task definitions

Policy PRM [11] first picks the highest pending job based on the period, similar to NP-RM, but also checks if we can still execute the next job of the highest priority task τ_1 after the highest priority pending job τ_i without causing a deadline miss for τ_1. We verify this by checking if τ_i can finish its execution before the latest start time of the next instance of τ_1, so we can guarantee the execution of τ_1.

As an example, consider the same taskset as Figure 3.1b and see Figure 3.3a at $t = 4$ after the execution of τ_2. PRM will first select the highest priority pending task τ_3 (line 2 in Algorithm 1). Next we calculate the deadline of the next instance of τ_1, with Equation 3.1, which gives us $D_1^{next} = 10$.

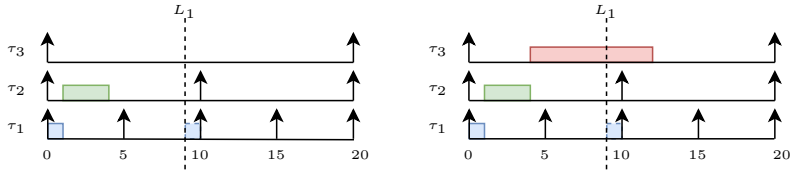
$$D_1^{next}(t) = (\lceil \frac{t}{T_1} \rceil + 1) * T_1 \tag{3.1}$$

Next we calculate the latest start time, at which τ_1 can start. Using Equation 3.2 this gives us $L_1 = 9$.

$$L_1 = D_1^{next}(t) - C_1 \tag{3.2}$$

Finally we verify if our highest priority pending job, τ_3, can finish its execution with Equation 3.3. If this equation doesn't hold, PRM will insert an idle-time. This is the case in our example since $4+8 \leq 9$ is *false*, visualized in Figure 3.3b. The final schedule is shown in Figure 3.4

$$t + C_i \leq L_1 \tag{3.3}$$



(a) τ_2 can finish before L_1 (b) τ_3 cannot finish before L_1

Figure 3.3: IIP for PRM

Algorithm 1: PRM [11]

Input: t: current time
Result: Idle-time decision for PRM
$\tau_i \leftarrow$ the highest priority task with a pending job;
$L_1 \leftarrow$ latest start time of the next job of τ_1 after t (Eq 3.2);
if $(t + C_i \leq L_1)$ ***or***
$(t + C_i \leq L_1 + T_1 - C_1$ ***and*** *τ_1 is the latest executed task)* **then**
  Dispatch τ_i;
else
  Insert idle-time from t to arrival of next job of τ_1;

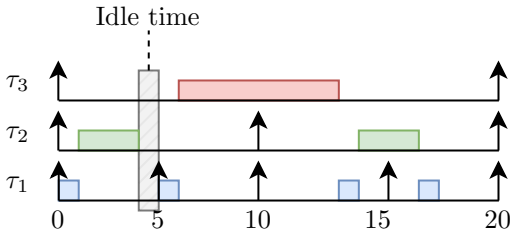


Figure 3.4: Final schedule with PRM

Shortcomings PRM also has its limitations. If we add τ_2, which is a new task with the same period as τ_1, to the task set (as seen in Figure 3.5) we notice that PRM results in a deadline miss for τ_2. τ_4 is still able to finish without causing a deadline miss for τ_1, so PRM doesn't insert an idle-time. But as a result, PRM violate the timing contraits of τ_2 in the schedule. It would be better to check if τ_2 can also finish before its deadline, together with τ_1.

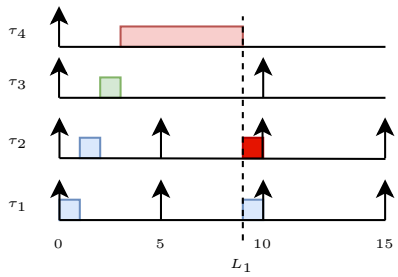


Figure 3.5: Shortcoming of PRM

3.3.2 Critical Window Earliest Deadline First

CW-EDF [10] is a scheduling policy which takes all tasks into account to check for any deadline misses, compared to PRM.

Policy The algorithm starts with selecting the highest priority job, which is based on the earliest absolute deadline. Then it will verify if scheduling this job will not cause any deadline misses for the next job of all non-pending tasks.

The main difference between PRM and this policy is that now all non-pending jobs are evaluated for deadline misses. We can see an example of such a calculation in Figure 3.6. In this example we have the situation whether τ_5 needs to be dispatched or not, since it is the only pending task in the system and all other tasks are non-pending. The policy first acquires the deadline of the next job for each non-pending task τ_k, using Equation 3.4.

$$D_k^{next}(t) = (\lceil \frac{t}{T_k} \rceil + 1) * T_k \tag{3.4}$$

Next we sort the k tasks based on their deadline in ascending order and calculate the latest start time of the last job in the list with Equation 5.1.

$$L_k(t) = D_k^{next}(t) - C_k \tag{3.5}$$

Then we iterate over the list in descending order (from k to 1) with Eq 3.6 to find the latest start time of all the other jobs. Note that we take the minimum value between the deadline and latest start time, because the execution of two tasks cannot overlap.

$$L_{k-1}(t) = min(D_{k-1}^{next}(t), L_k(t)) - C_{k-1} \tag{3.6}$$

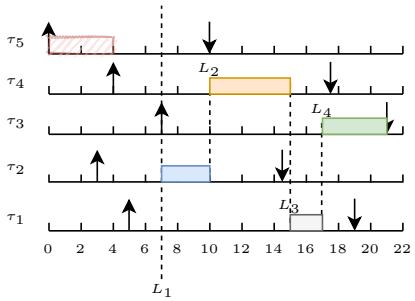
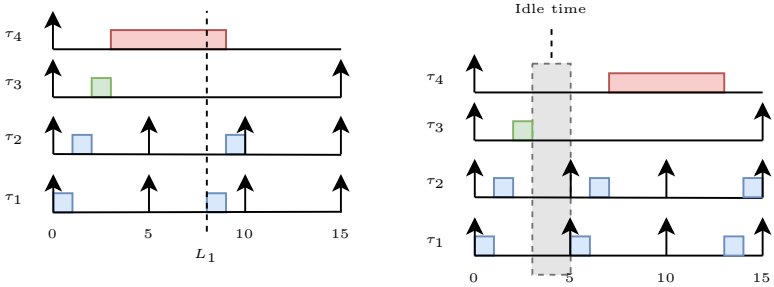


Figure 3.6: Calculating L_1 with multiple jobs

Example We use the same taskset as in Figure 3.5. We noticed that the execution of τ_4 causes a deadline miss for τ_2 with PRM. Now with CW-EDF at $t = 4$ (Figure 3.7) we check if scheduling τ_4 will result in a deadline miss for any next job of the non-pending tasks by calculating their latest start times. We see that L_1 is 8, where $t + C_4$ is 9, which will result in an idle-time insertion. The algorithm is shown in Algorithm 2.



(a) τ_4 cannot finish before L_1

(b) Final schedule

Figure 3.7: CW-EDF

Algorithm 2: CW-EDF [10]

Result: Idle-time decision for CW-EDF

```
t ← current time;
τ_i ← the pending job with the earliest deadline;
Obtain L_1(t) from Equation 3.6;
if t + C_i ≤ L_1(t) then
    Dispatch job τ_i;
else
    Insert idle-time from t until release of critical job;
```

We see that by adding all non-pending jobs into the calculation of the latest start time we can find schedules without deadlines misses where policies like

NP-RM and PRM failed to do so. There is however a major downside to CW-EDF, the overhead. NP-RM only has to find the job with the shortest period and additionally with PRM, we only have to calculate the latest start time of one task. But with CW-EDF we need to sort all non-pending jobs by their deadline first and then iterate over those jobs to find the latest start time of all jobs.

This leads to the desire to have a policy which performs just as good as CW-EDF but with a lower overhead. We need to detect and skip calculations that are being done which have no effect on the scheduling decision.

3.3.3 Event-driven policies

Current work on event-driven systems rely on preemptive systems in order to cope with event-triggered tasks [15] or use a distributed system [12]. Current policies, as described in this section, are bounded by non-preemptive systems and deal with event-triggered tasks using the lowest period measured. This sets the deadline of the next job, which is used in PRM and CW-EDF, to an earlier time than the actual arrival. This thesis will investigate the ability of finding less pessimistic deadlines to improve schedulability.

3.4 Summary

We've shown several existing state of the art non-preemptive non-work-conserving policies. We see that PRM [11] can improve schedulability by inserting idle-time with very low overhead by verifying that the to be dispatched task doesn't cause a deadline miss for the highest priority task. In order to increase the schedulability of PRM we showed CW-EDF [10], which checks all non-pending tasks for deadline misses. While this improves schedulability, this also increases the overhead polynomially. Above policies requires knowledge of the future and are designed for non-preemptive periodic tasks. Policies for event-triggered tasks where this knowledge is absent are still in the works. Because of this, current policies can be too pessimistic regarding the next arrival time and have a poor schedulability compared to periodic task sets. In this thesis we will investigate the field of creating a policy which is designed to cope with event-triggered tasks.

Chapter 4

System Model and Problem Definition

In this chapter we define the model for both time and event-triggered systems. Our problem definitions are described which will be answered in Chapters 5 and 6. Finally our assumptions are described which will bound our model.

4.1 Time-triggered system model

We are given a task set $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$, where n is the number of tasks in the set. Each task is identified by $\tau_i = (C_i, T_i, D_i)$, where $C_i, D_i, T_i \in \mathbb{N}$.
C_i is the execution time of the task, T_i is the period and D_i is the relative deadline where $D_i \leq T_i$. Properties that can be derived from the task set:

- Utilization: $U_i = C_i/T_i$.
- Hyperperiod: H. The least common multiplier (LCM) of the period of all tasks.

Each job is denoted by $J_{i,j}$, where j represents the jth instance of the task. Each job has a release time $r_{i,j} = j * T_i$ and an absolute deadline $d_{i,j} = r_{i,j} + D_i$. Task τ_i is visualized ins Figure 4.1.

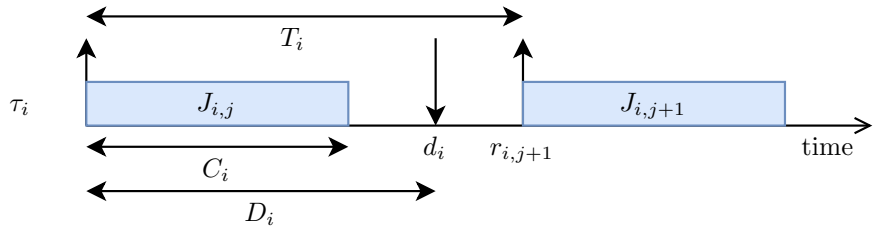


Figure 4.1: Visualization of task τ_i

Problem Definition Given a task set τ, can we find a schedule until the hyperperiod with no deadline misses, with a lower overhead than CW-EDF?

4.2 Event-triggered system model

We are given a task set $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$, where n is the number of tasks in the set. Each task is identified by $\tau_i : (C_i, D_i, A_i)$, where $C_i, T_i \in \mathbb{N}$.
C_i and D_i are the same as the time-triggered model. A_i is the arrival curve of the task. The history size is denoted by M.

Arrival curve A_i is defined as a table (Table 4.1). Where the table is constructed as follows: $A_i^{j_{min}}$ denotes the minimal interval between j consecutive jobs of task i. $A_i^{j_{max}}$ denotes the maximum interval between j consecutive jobs of task i. The table is visualized in Figure 4.2.

Number of consecutive events	Minimum interval	Maximum interval
2	$A_i^{2_{min}}$	$A_i^{2_{max}}$
j	$A_i^{j_{min}}$	$A_i^{j_{max}}$
$j+1$	$A_i^{j+1_{min}}$	$A_i^{j+1_{max}}$
$\vdots$	$\vdots$	$\vdots$
M	$A_i^{M_{min}}$	$A_i^{M_{max}}$

Table 4.1: Arrival curve table of A_i

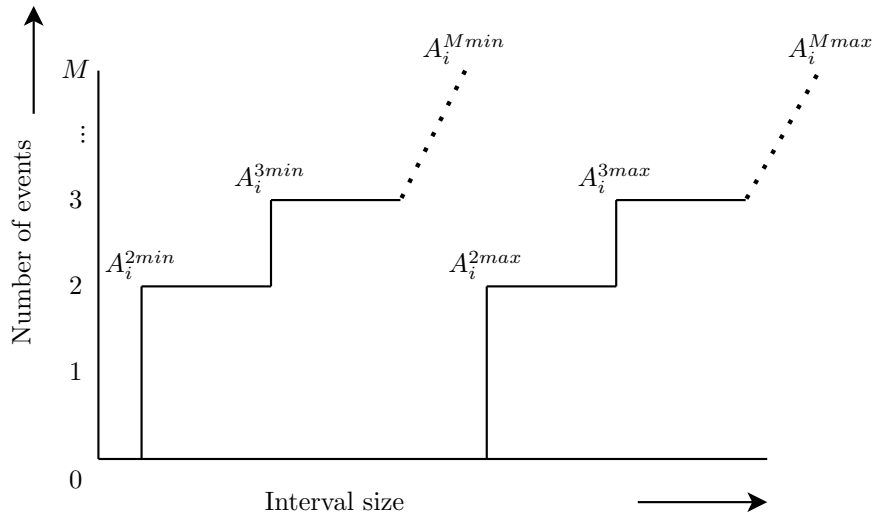


Figure 4.2: Visualization of arrival curve

Problem Definition Given a task set τ, can we find a schedule with lower number of deadline misses than CW-EDF?

4.3 Assumptions

We assume independent tasks on an uniprocessor without shared resources. Tasks are indexed by there period so that $T_1 \leq T_2 \leq ... \leq T_n$. For time-triggered

systems we assume a hard real-time execution model and for event-triggered systems we assume a soft real-time execution model. For all values of C_i we assume that this is the Worst-Case Execution Time (Worst-Case Execution Time (WCET)). If the job finishes earlier, the task will busy-wait until the WCET is reached.

Chapter 5

An Efficient Scheduling Policy for Time-Triggered Systems

In this chapter we introduce an efficient scheduling policy for time-triggered systems. Our goal is to match the schedulability of CW-EDF with a reduced overhead.

5.1 Motivation

In previous chapters we mentioned two non-work conserving policies, PRM and CW-EDF. Nasri et al. [11] showed a significant increase of schedulability with the introduced IIP, while adding only a constant overhead $O(1)$. There we found that adding a similar task as τ_1, it can cause a deadline miss for the added task. This was schedulable by CW-EDF, but the major downside is its overhead $O(n \log n)$ as shown by Nasri et al.[9]. Because CW-EDF takes all non-pendings tasks in consideration, CW-EDF might be taking tasks into consideration which are not critical for its idle-time insertion decision. We will detect these critical tasks before running new policy with the help of CW-EDF by verifying for each idle-time insertion decision which tasks were of influence for this decision. During our new policy, we will only take these critical tasks into consideration. This will lead to an overhead which is at best the same as PRM and at most the same as CW-EDF, while remaining the schedulability of CW-EDF. We call our policy K-Precautious Earliest Deadline First (KP-EDF).

5.2 KP-EDF

In order to reduce the overhead of CW-EDF, we will try to find tasks that have no influence on the idle-time insertion policy. If we skip those tasks during the calculation of the latest start time (L_1), we will have the same outcome as CW-EDF, but with a lower overhead. To do this, we will find the critical tasks of each idle-time insertion of CW-EDF. KP-EDF will run CW-EDF prior to running the system to find the critical tasks and will store this in a small table

used by KP-EDF. The policy of KP-EDF is equal to CW-EDF after finding these critical tasks.

5.2.1 Critical tasks of a CW-EDF decision

We start with an example of an idle-time decision of CW-EDF. By identifying the critical tasks in CW-EDF, we will only use these tasks for the IIP of KP-EDF. By having less tasks to evaluate in our IIP, we reduce the overhead.
Consider the following tasks with their deadlines in Table 5.1 at $t = 100$. We have one pending job τ_6 and CW-EDF will calculate the latest start time of all non-pending tasks, which are all other tasks. We do this with Equation 5.1 and 5.2, as explained in Section 3.3.2 for all non-pending tasks. CW-EDF will look into the future and check if it can schedule τ_6 without causing a deadline miss for any of the other tasks. In this example scheduling τ_6 will cause a deadline miss because τ_6 will finish (107) after the latest start time (106), Equation 5.3 will be false, hence CW-EDF will insert an idle-time. $L_1(t)$ in Equation 5.3 respresents the latest start time of the non-pending tasks.

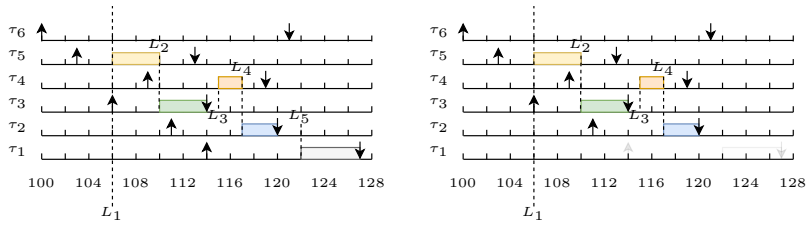
$$L_k(t) = D_k^{next}(t) - C_k \tag{5.1}$$

$$L_{k-1}(t) = min(D_{k-1}^{next}(t), L_k(t)) - C_{k-1} \tag{5.2}$$

$$t + C_i \leq L_1(t) \tag{5.3}$$

τ_i	C_i	d_i
τ_1	5	127
τ_2	3	120
τ_3	4	114
τ_4	2	119
τ_5	4	113
τ_6	7	121

Table 5.1: Tasks with their computational time and absolute deadline

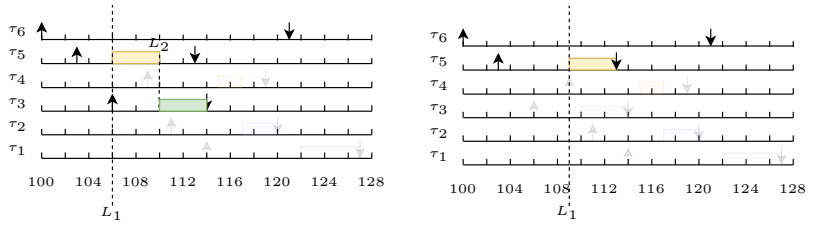


(a) All non-pending tasks (b) Ignoring τ_1

Figure 5.1: Calculating latest start time ($L_1(t) = 106$)

Non-critical tasks CW-EDF takes all the non-pending tasks into consideration in order to find the latest start time, which is 106. The trick is to check if there are tasks that can be skipped in calculating the latest start time, while the idle-time decision stays the same. In other words, we try to identify the tasks which are critical for determining this idle-time decision, since having the same decision as CW-EDF will result in an policy with equal schedulability. We start by ignoring the task with the latest deadline of next instance of the non-pending tasks, which is τ_1. We see in Figure 5.1b that ignoring τ_1 has no influence on the latest start time, thus not changing the idle-time decision. Therefore, we can conclude that τ_1 is not critical for this idle-time decision. We repeat this process of ignoring additions tasks (based on the latest deadline) until the outcome has no idle-time insertion. We see that if we continue this process we can also ignore τ_2 and τ_4 while still having the same idle-time insertion, as seen in Figure 5.2a.

Critical tasks We continue this process until we have no idle-time insertion. This happens if we also ignore τ_3, as seen in Figure 5.2b. The latest start time is different (109) and due to Eq. 5.2 we will actually cause a different scheduling decision. τ_6 will be dispatched because it can finish before the latest start time which is now 109. The example in Figure 5.2 shows that τ_3 and τ_5 are critical for this idle-time decision. KP-EDF can ignore τ_1, τ_2 and τ_4 to calculate the latest start time, while still getting the same outcome for Equation 5.2, leading to the same idle-time decision.



(a) Continuing ignoring tasks (b) Ignoring too many tasks

Figure 5.2: Different idle-time decision

Pushing effect Looking at the latest start time of τ_3 and τ_5 in Figure 5.2a we see that τ_3 *pushes* τ_5 forward. This gives us some indication that tasks might be critical. If non of the tasks are pushing, we can safely ignore all the other tasks except the first one, because this task causes the idle-time insertion for CW-EDF. We call this phenomenon the pushing effect.

In this example we've seen that τ_3 and τ_5 are critical for this idle-time decision. We were able to remove τ_1, τ_2 and τ_4 from the IIP calculation, while still remaining the same decision. Therefore, these tasks were not critical for this decision.

5.2.2 Finding all K critical tasks

Finding the critical tasks is an offline process, after which the critical tasks are stored in a table and used by KP-EDF. During this process, we will simulate

one hyperperiod with CW-EDF and find critical tasks each time CW-EDF encounters an idle-time insertion. KP-EDF will take the non-pending tasks and iteratively remove the task with the latest deadline while maintaining the same idle-time decision. Tasks that remain in this list are considered critical. The removed tasks are considered non-critical, but *only* for this decision. Non-critical tasks for one idle-time decision can be critical for another idle-time decision.

On each idle-time decision KP-EDF will sum the critical tasks of *all* idle-time decisions of CW-EDF. The critical tasks found in all those decisions will be added to the final critical task set. KP-EDF will consider all found tasks in each idle-time decision of CW-EDF as the critical tasks.

Algorithm The algorithm to detect the critical tasks is shown in Algorithm 3. To find all critical tasks, KP-EDF will first run CW-EDF for one hyperperiod (line 1). Each time CW-EDF encounters an idle-time decision (line 2) it will use the jobs and tasks used in the decision. It will start by fetching the highest priority job (line 3) and the non-pending tasks (line 4). Next we sort the next jobs of the non-pending tasks by their deadlines (line 5) and remove the job with the latest deadline (line 6). Then we calculate the latest start time of the leftover jobs (line 7) and check if this leads to another idle-time decision (line 8). We keep removing the task with the latest deadline (line 9) until the line 8 isn't true anymore. We add the remaining tasks to the final critical task set until we reach the hyperperiod. At the end *all* critical tasks found in *all* idle-time decisions are be added to the final set of critical tasks.

Algorithm 3: Finding critical tasks

```
   Data: C: Critical tasks, t: time
   Result: Return the critical tasks as set C
 1 while Running CW-EDF do
 2     if CW-EDF insert idle time then
 3         τ_i ← highest priority pending job;
 4         P ← non-pending tasks;
 5         Sort P by deadline in ascending order;
 6         Remove the task with the latest deadline from P;
 7         L_1(t) ← latest start time of tasks in P (Equation 5.2);
 8         while t + C_i > L_1(t) do
 9             Remove task from with the latest deadline from P;
10             L_1(t) ← latest start time of tasks in P (Equation 5.2);
11         end
12         Add tasks in P to C if not already in C;
13 end
```

5.2.3 Policy

KP-EDF works the same as CW-EDF, after defining the critical tasks. For each decision the scheduler chooses the highest priority pending job based on the deadline to check if it can be scheduled. Next, it verifies if scheduling this job

will cause a deadline miss for the next job of the critical tasks. If scheduling the highest priority pending job doesn't cause any deadline misses for the critical tasks, it will be dispatched. An idle-time is inserted otherwise. The algorithm is shown in Algorithm 4.

Overhead Because KP-EDF has the same computational complexity as CW-EDF, but with less tasks (k), we can state that the overhead of KP-EDF is $O(k \log k)$ where $k \leq n$.

Algorithm 4: KP-EDF

```
   Data: J: Tasks, CT: Critical Tasks, HP: Hyperperiod
 1 t ← current time;
 2 CT ← Critical tasks of set J with Algorithm 3;
 3 while t ≤ HP do
 4 |   if a job is released and the processor is idle then
 5 |   |   τ_i ← pending task with earliest deadline ;
 6 |   |   L_1(t) ← get latest start time of critical tasks (CT);
 7 |   |   if L_1(t) ≤ time + C_i then
 8 |   |   |   insert idle time until the next critical job;
 9 |   |   else
10 |   |   |   dispatch(τ_i);
11 |   |   end
12 |   end
13 end
```

Chapter 6

A Scheduling Policy for Event-Triggered Systems

In this chapter we introduce a policy for event-triggered systems. We start with the foundation of the arrival curve, where we explain the generation first. Next we describe the usage of the arrival curve in the policy, which will focus on reducing the number of deadline misses in a soft real-time environment.

6.1 Arrival Curve

As explained in Section 2.1.1 the arrival curve represents the lower bound and upper bound on the inter-arrival time between consecutive events in a system. We show how we generate and can use this arrival curve, to help the policy estimate the future based on these measurements of the task.

6.1.1 Generating arrival curve from event traces

In order to construct the arrival curve, we apply the algorithm described in Algorithm 5 to the events measured. This algorithm finds the lower and upper bound on the inter-arrival time between a finite (noted by M) consecutive number of events based on a measurement of m time instants.
$T =< t_1, t_2, \ldots, t_m >$ is an ordered set of time instants t_i at which the event has happened during measurement of the task. The value M represents the height of the arrival curve. In theory M can be as high as m, but this requires an table with M rows, which doesn't fit into memory as we want as many measurements as possible. Since the policy will access each row of the table during its IIP, we need M to be as low as possible. A higher value of M will increase the overhead. The effect of the size of M is shown in Section 7.3.

Example Repeating the introduction of the arrival curve; assume we measured the arrival events from an event-triggered task at the following timestamps: 2, 10, 13, 15, 20, 24. We calculated that the minimum time between two arrivals is 2 (between 13 and 15) and the maximum is 8 (between 2 and 10). We can also perform this calculation for three consecutive events, we see that the minimum is 5 (between 10 and 15) and the maximum is 11 (between 2 and 13).

We can repeat this process for 4 and more consecutive events. This is visualized in Figure 6.1. We store this information in a table which is shown in Table 6.1. This table can also be visualized, as shown in Figure 6.2.

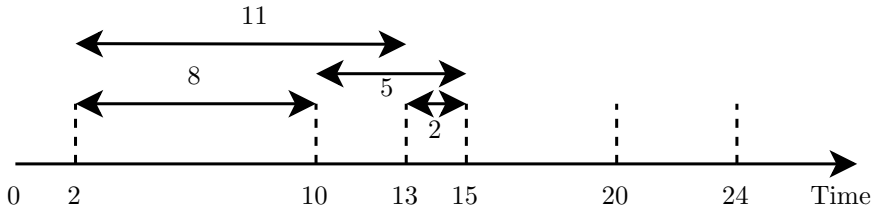


Figure 6.1: Intervals between consecutive events

# of consecutive events	Minimum interval	Maximum interval
2	2	8
3	5	11
4	10	14
⋮	⋮	⋮
M	A_i^{Mmin}	A_i^{Mmax}

Table 6.1: Example table with consecutive events

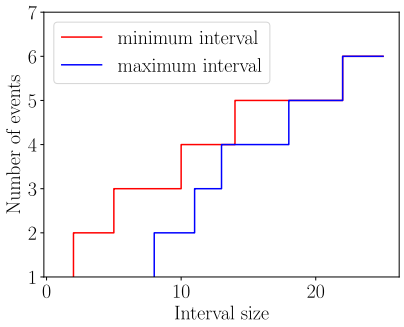


Figure 6.2: Arrival curve

6.2 Motivation

For all policies we've mentioned in this thesis so far, they all work for strictly periodic tasks. Event-triggered tasks don't have a static period and this causes issues for those policies. When determining the latest start time, the non-pending tasks use the release time of the next job to calculate the deadline of the future job. However, this period and thus release time is unknown and current policies use the lowest interval measured as the period instead. It is guaranteed that the next job will not arrive earlier than this lowest interval.

Algorithm 5: Computation the arrival curve

```
Input: T: Measured traces of events, H: Maximum height of arrival
       curve (number of consecutive events)
Result: Returns the lower and upper bound arrival curve
for h ≤ H do
    A^{h_max} ← 0
    A^{h_min} ← ∞
    for i ≤ length of T do
        δ ← T_{i+h} − T_i
        A^{h_max} ← max(A^{h_max}, δ)
        A^{h_min} ← min(A^{h_min}, δ)
    end
end
```

Using this value is influencing the calculation of the latest start time of the other tasks. It is possible that the expected arrival would be a critical task for the idle-time decision, but the actual arrival (which is unknown) will be considered a non-critical task. This might cause incorrect idle-time decisions and can cause deadline misses. Therefore we need a policy which uses the arrival curve to estimate the behavior of the future jobs, by using the arrival curve. We call this policy Arrival Curve EDF (ArC-EDF).

6.3 ArC-EDF

In this section we will present a policy which will try to find a schedule for systems with event-triggered tasks that lead to the least number of deadline misses. We present a policy which uses the arrival curve of an event-triggered task to calculate the latest start time of the non-pending tasks, using an estimated value of the next arrival for event-triggered tasks. We can use different values on the arrival curve anywhere between the minimum and maximum arrival time. An explanation of possible values is in Section 6.4. Examples of values to take are the minimum, maximum and average inter-arrival timings.

6.3.1 Policy

ArC-EDF keeps a history of arrivals of the event-triggered task. The policy then uses this history to see how the system behaved in the past and uses the arrival curve to calculate how it might behave in the future.

Usage As stated, ArC-EDF will use the history and arrival curve to estimate the deadline of the next instance of an event-triggered task. This value is then used as release time for the calculation of the latest start time.

Example For this example we will use the event-triggered task and the calculated arrival curve as in Section 6.1.1.
We denote the history of the task as $H =< h_1, h_2, \ldots, h_M >$ with M being our history size and height of the arrival curve. The history is a stack of size M. A

new arrival is pushed onto the stack at h_1 and h_m is pushed off. Consider that during execution task τ_i has arrivals on the following timestamps: 8, 12, 19. We now want to know the arrival of the next job. More specifically, we want to know the deadline of the next job, since this value is used in the calculation of the latest start time. Since we have 3 events in our history, we will look for the arrival of the next jobs using our arrival curve with those 3 timestamps. For each event in our history H we will calculate the expected arrival with Equation 6.1, where j is the position in our history. For the latest arrival in our history ($h_1 = 19$) we will look 1 event into the future. For the second arrival ($h_2 = 12$) in our history we will look 2 events into the future, and so on. Eventually, after calculating the next arrival of all the events in the history, we select the maximum value of these next arrivals as the release time of our next job (Equation 6.2). We take the maximum value because all estimated arrivals guarantee that the arrival won't be before their estimation based on the measurements. Please note that in Equation 6.1 we don't define if we use the minimum inter-arrival time ($A_i^{j_{min}}$) or the maximum inter-arrival time ($A_i^{j_{max}}$) as presented in Table 6.1. In Section 6.4 we discuss this selection.

$$r_i^j = h_j + A_i^{j+1} \tag{6.1}$$

$$r_i = \max_{\forall j \in M} r_i^j \tag{6.2}$$

For our example we use $A_i^{j_{min}}$. The results are in Table 6.2 and visualized in Figure 6.3, where the dotted lines are our events in the history and the up-arrows are the possible next arrival based on the arrival curve. Using Equation 6.2 we get a r_i of 21 which is our estimated next arrival.

h_j	$A_i^{j+1_{min}}$	r_i^j
19	2	21
12	5	17
8	10	18

Table 6.2: Results of Equation 6.1

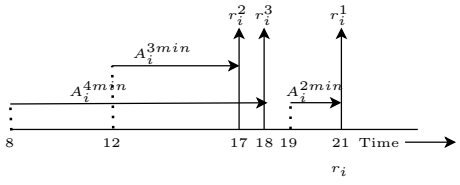


Figure 6.3: Estimated arrivals of the next instance of τ_i

Final algorithm

The ArC-EDF policy is identical to CW-EDF. ArC-EDF select the highest priority pending job and will evaluate the latest start time to insert an idle-time decision. The difference is the additional calculations of the event-triggered

Algorithm 6: Finding lower bound

```
Data: M: History Size, H: History
r_i ← 0;
j ← 1;
while j ≤ M do
    r_i^j ← h_j + A_i^{j+1} (Eq. 6.1);
    r_i ← max(r_i, r_i^j) (Eq. 6.2);
end
```

tasks regarding the deadline and latest start time. In fact, if we use a history size of 1 and use the minimal inter-arrival time, we have CW-EDF. This is because the minimal inter-arrival time of two consecutive events is the value CW-EDF uses for calculating the latest start time.

Overhead Since CW-EDF has a time-complexity of $O(n \log n)$, but we add M (history size) calculations to every event-triggered task, the time-complexity of KP-EDF is $O(n \log n * M)$.

Algorithm 7: ArC-EDF

```
Data: t: current time, HP: Hyperperiod
while t ≤ HP do
    if a job is released and the processor is idle then
        τ_i ← highest priority pending job;
        PJ ← non-pending jobs ;
        Sort PJ on deadline;
        L_1(t) ← latest start time of tasks in PJ;
        if t + C_i ≤ L_1(t) then
            Dispatch job τ_i;
        else
            Insert idle time;
        end
    end
end
```

6.4 Value used on the arrival curve

While the example only accesses the arrival curve for the minimum inter-arrival time, there is more data to be used in the arrival curve. We could pick an interval anywhere on the horizontal line between the maximum and minimum interval as the inter-arrival time to calculate the latest start time for our policy. Using the minimum interval would imply that we guarantee that the actual arrival is later, while selecting the maximum interval would ensure that the next job will certainly arrive before this point. Note that we cannot use both the minimum and maximum inter-arrival time during the policy, so we have to choose this from the start.
In Section 7.3 we evaluate different intervals picked between the minimum and

maximum inter-arrival time, based on the number of tasks and utilization of the system and how this effects the number of deadline misses. In short, the next arrival could be anywhere between point a and d for two consecutive events (Figure 6.4) so we could pick any value on this horizontal line as our estimated period. The same applies for three consecutive events between point c and f. So there is a difference between ArC-EDF which uses the maximum inter-arrival time and ArC-EDF which uses the minimum inter-arrival time and these can be seen as two different policies.

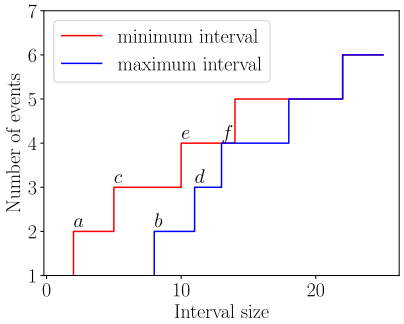


Figure 6.4: Arrival curve of an event-triggered task

Chapter 7

Evaluation

In this chapter we will evaluate our designed policies from Chapter 5 and 6. First we will perform experiments to measure if we can **lower the overhead of CW-EDF**, addressing **RQ1**. Next we measure the ability to **reduce deadline misses in event-triggered systems**, addressing **RQ2**.

In order to answer these questions we used two frameworks. The first framework is created in Java and generates, simulates and shows the results regarding schedulability and deadline misses. The second framework is inspired by an existing framework for Arduino to measure the overhead. With these two framework we measure the defined performance metrics in Section 2.2.2.

7.1 Evaluation framework

The implementation of the solution is done using two frameworks. All simulation runs are performed using the self-made Java framework. The run-time overhead is measured using an Arduino framework on actual hardware, an Arduino Mega. The Java framework is able to generate, run and visualize all task sets and policies. The Java framework also generates the files that will be used on the Arduino for run-time measurements.

7.1.1 Simulation

Configuration

The configuration file consist of multiple settings which sets the environment for the simulator.
The file consists of the following setting arrays:

Taskset types
Utilizations
Number of tasks
Policies
Amount of samples for each configuration

Generation

Given a configuration, the framework will first generate random task sets. This is done using simple for-loops on the variables in the configuration. The framework will now generate all possible combinations of the configuration with a given sample size. All generated task sets are then saved to a storage device. Saving the task sets to disk gives us the opportunity to rerun policies without having to generate the task sets again, which can be time consuming.

Basic requirements For every taskset we have several requirements in order to be a valid task set for our simulation.

Max C The first rule is $\forall i C_i \leq 2 * (T_1 - C_1)$. This states that the all values of C_i have to be smaller than the twice the slack of the job with the smallest period. As seen in figure 7.1, the maximum amount of time between the two jobs of τ_1 is twice its slack. If the computational time of the second task is larger than this, it will cause a deadline miss for τ_1. If we encounter such a value during our simulation, we dismiss the task set and generate a new one until we have enough task sets.

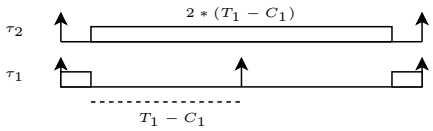


Figure 7.1: Max computational time

Maximum number of jobs This rule is mainly done to keep the simulation runtime within certain boundaries. Since the simulator will run until the hyperperiod of each task set, we cannot allow it to have a very large hyperperiod. For instance, a few tasks with a period of 10, 50 and 100 will have a hyperperiod of 100. But a task set with similar periods such as 11, 53 and 101 will have a hyperperiod of 58883. Since both task sets are similar, we would rather use the first one because it's faster to simulate. For this reason we would like each job to have a maximum amount of jobs until the hyperperiod. This keeps the tasks periods within a reasonable range of each other. While a taskset with tasks of a period of 5 and 2,000,000,000 will be able to be simulated, the use case is not realistic and the generation tool will have a chance to generate these type of task sets. This example will run 400.000.000 jobs before reaching the hyperperiod. Our maximum value of jobs until the hyperperiod is therefore set to 100000.

Loading After generating the sets, the framework will load them from the disk. The framework will not load all task sets into memory immediately, but keep it above a certain threshold. If the amount of loaded task sets drops below this threshold, more are loaded into memory. When a task set is loaded into memory, it gets cloned for each policy that we will simulate.

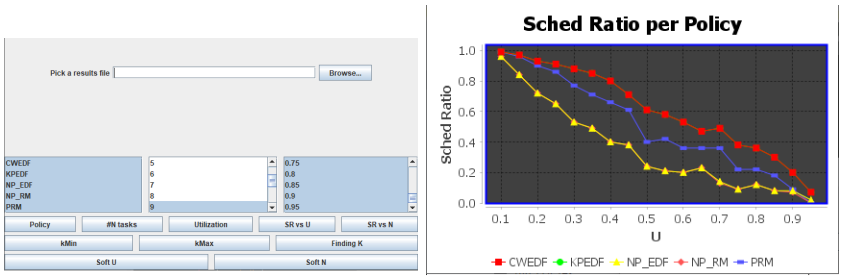
Simulation

Since each policy will run a task set and is independent on other resources, the framework uses multiple threads to run each simulation. This speeds up the simulation time by a factor of the amount of cores on the system. A simulation will stop when a deadline miss occurs or it reaches the hyperperiod. Results are then saved to disk.

HPC For the final results we used the High Performance Cluster located at Delft University of Technology. Using this cluster we were able to get a large amount of cores and memory for our simulations (eg 64 cores and 32GB of RAM), allowing several million of task sets can be simulated within a few hours. Result files were downloaded from the clusters and inspected on a local device, using our visualization tool.

Visualization

The Java framework can also generate the graphs of the simulation after execution. The results file are loaded into a separate window and the user can show different types of graphs, given a certain configuration. This tool was mainly used during development and debugging.



(a) Result form to generate graphs

(b) Graph for selected values by the form

Figure 7.2: Schedulability ratio of log uniform task sets

Simulation results The graphs used in this thesis are not generated by the framework, but are generated using matplotlib [6] in Python. The reason for this is the flexibility of matplotlib and the ability to produce SVG images.

Individual schedules In order to debug a policy for any given task set, the framework can show individual schedules. An example is shown in figure 7.3

7.1.2 Hardware implementation

We created an simulation framework inspired by Nasri et al. [9]. We implemented the KP-EDF and ArC-EDF policies to measure our work, but also used other policies already implemented to compare our work against existing policies.

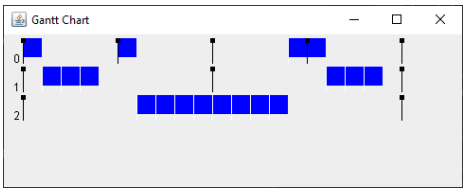


Figure 7.3: Visualization of a single schedule

Generation In order to generate the files needed on the SD Card, we run the Java framework with a given configuration and store all task sets to the SD card with the correct syntax. Only task sets that didn't have any deadline misses during simulation are used, so the framework will continue to generate and simulate task sets until the required number of samples is reached.

7.2 Evaluation on time-triggered systems

Baselines We will compare our policy against the existing state of the art policies PRM and CW-EDF. We will measure the difference in overhead. For this we use the Arduino platform to measure the overhead of task sets for each policy. We will also measure the size of the critical task set with respect to the number of tasks, which gives us a theoretical gain compared to CW-EDF.

7.2.1 Experimental setup

Generation For our experiments, we generate different types of task sets and evaluate them on different metrics. In this experimental setup we use two methods to generate task sets, namely log uniform [4] and automotive [8]. Log uniform task sets have tasks where the periods are generated uniformly over a log scale. Automotive task sets are generated using real life periods used in the automobile industry.

Log uniform task sets Using a log uniform generator, it will generate periods that are distributed over a logarithmic scale. We use the generation method described by Emberson et al. [4]. Figure 7.4 gives a visual representation of what a selection might look like.

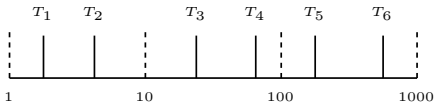


Figure 7.4: Log uniform period selection

Periods To generate these periods, the generator has 3 variables: lower-bound, upper-bound and base. Each period generated will be dividable by the base. Having a large value will be in favor of the hyperperiod, since the generated periods will have a lower Least Common Multiplier (LCM). But, there are less

periods available to be generated. To keep the generated periods within a certain boundary we will also set the upper and lower bounds from which the generated periods will be selected. Having a small base and lower bound, but also a high upper bound will lead to a large selection of different periods. This also leads to high hyperperiods, which are quickly too high for our maximum hyperperiod. Therefore, we have to set the parameters to a value which will lead to a large amount of different periods, while still have a hyperperiod below our set value. Our generated log uniform test set has a base of 100, lower bound of 100 and an upper bound of 6000. This gives us 60 different periods to be selected.

Execution time With the periods generated, the execution times must also be generated at random. It is not possible to generate the execution by a fraction of the period, since all tasks will then hold the same utilization. The method described in the work of Emberson et al. [4] uses UUnifast. This method will generate a random number of values, that will sum up to a given value. So if we would need a task set with a utilization of 0.8 and 8 tasks, then UUnifast will generate 8 float values that add up to 0.8. If we multiply these float values with their respective period, the result is a task set with a utilization of 0.8, 8 tasks and randomly selected execution times.

Automotive task sets Using this method we will generate a task set which will use the periods used in the automotive industry, introduced by Kramer et al [8]. This is a standard that is used across many manufactures. Testing such task sets will test our policy with a real life scenario of task sets. These samples are called runnables. Our idea is to pick runnables until we reach a randomly selected utilization for our task set. Next we merge these runnables into tasks until a random selected execution time is reached. Because we combine the runnables based on the utilization, we have no control over the number of tasks being generated.
The algorithm starts with a given utilization. It draws runnables with a given utilization and keeps doing this until the final utilization is reached with all the runnables. Next it merges the runnables with the smallest period together and generates the task. From here we can retrieve the maximum value for our execution time, as explained in section 7.1.1. From here on, the algorithm will randomly select a value between 1 and the maximum execution time. With this value we will merge runnables with the same period until this execution time is reached. If we have no more runnables left, we will generate a task with the current merged runnables. This process repeats until all the runnables are merged into a task. Because we generate the tasks with a randomly selected value of U_i, we have no control over the number of tasks generated. Because of this we only evaluate the policy based on different value of U.

Design of the experiment

For our experiments we generate 1000 task sets for each data point and vary the following parameters:

- **Number of tasks** We vary the number of tasks to measure the behavior of our policies with respect to n. We consider the following number of tasks: [2...15].

- **Utilization** We vary the utilization of the task sets as a fraction of how busy the system is. We consider the following values: [0.1, ..., 0.95] with intermediate steps of 0.05.

For our overhead experiments we generated 250 task sets and let each policy run for 30 seconds on the Arduino platform. We generated log uniform task sets with the number of tasks of [3, 6, 9, 12] and random utilizations in the range of [0.50, 0.95].

In the first experiment, we measure the overhead on a real embedded hardware platform, the Arduino Mega. The second and third experiment will measure the calculated critical task set size, compared to the maximum number of tasks CW-EDF encountered to generate the latest start time. For these experiments we will vary in both number of tasks and utilization. We deliberately didn't use a box plot to visualize the critical size set of KP-EDF since the average values we so low the image became very cluttered and confusing. For this reason we only show the maximum and average critical set size.

7.2.2 Results

The results for our new policy KP-EDF show a significant drop in overhead. The number of critical tasks in a task set is relatively to the number of tasks very low (average fraction of 20%). The first experiment shows that the actual overhead measures on an Arduino is significantly dropped, up to 60%. The other experiments show the the size of the critical task set is small compared to the number of tasks CW-EDF evaluates for the latest start time.

Impact of the number of tasks on the overhead In this experiment we look into the effects of the number of tasks on the overhead. We've taken multiple policies and let them run with different number of tasks on an Arduino platform. The utilization varies randomly between 0.50 and 0.95. We ran 250 task sets for each number of tasks (3,6,9,12). The result is shown in Figure 7.5. We see that the number of tasks has a large effect on CW-EDF, where KP-EDF increases less. For 12 tasks we see that the average overhead of CW-EDF is decreased with 60% compared to KP-EDF, while the schedulability remains equal. This indicates that CW-EDF takes a lot of non-critical tasks into consideration for the latest start time. We also see that this causes a significant variation in the overhead of CW-EDF.

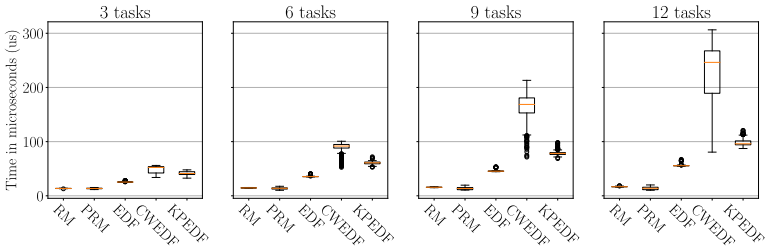


Figure 7.5: Impact of the number of tasks on the overhead

Impact of the number of tasks on the size of the critical task set This experiment will show the size of the maximum number of tasks evaluated during the calculation of the latest start time with respect to the number of tasks in the system, all with a utilization of 0.7. In Figure 7.6 we see that the critical task set size is very small compared to the total number of tasks in the system. For instance, with a task set of 15 tasks and a utilization of 0.9 we see that the average critical task set size is only 1.9.

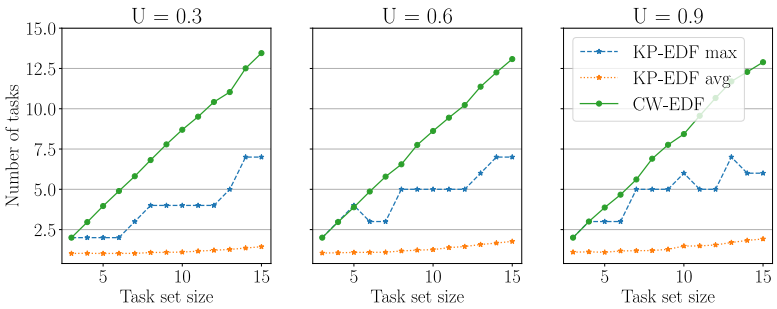


Figure 7.6: Impact of the number of tasks on critical task set size

Impact of the utilization on the size of the critical task set Here, we look into the impact of the utilization of the system, for various task set sizes. We see that the influence of the utilization is low. One interesting observation is that the maximum number of tasks considered by CW-EDF decreases with an increasing utilization. This is due to the fact that a system with a high utilization will have more jobs in the ready queue, while we only calculate with the non-pending jobs.

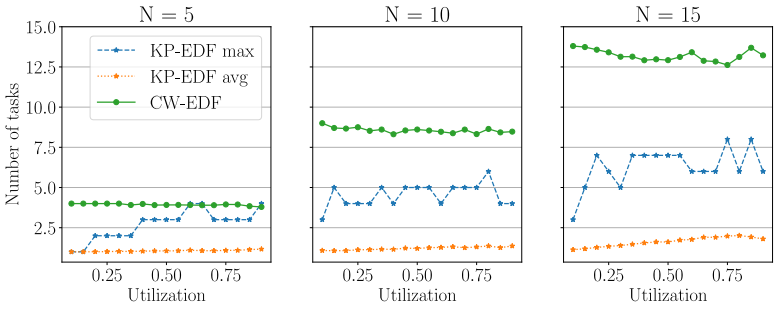


Figure 7.7: Impact of the utilization on critical task set size

Critical task set size for automotive task sets An interesting finding was that the critical task set size for automotive task sets was *always* 1, no matter the utilization or number of tasks. In the works of Nasri et al. [10] we see that CW-EDF outperforms PRM when the period ratio's between the tasks are low (underneath 2). With automotive taskset we generate one task with the lowest period, so the period ratio with the second task is 2.

7.3 Evaluation on event-triggered systems

Baselines In these experiments we will compare the deadlines misses with Precautious-RM [11]. For our own policy we will inspect the number of deadline misses based on the horizontal selected value on the arrival curve, as described in Section 6.4. Keep in mind that ArC-EDF with a history size of 1 and using the minimum inter-arrival time is effectively the same as CW-EDF. To measure the deadline misses we use our Java framework. Every taskset will be running for 50 hyperperiods, where the hyperperiod is calculated by using the minimal interval of the event-triggered tasks as the period.

7.3.1 Experimental setup

Generation We generate the tasks in a similar fashion as the log uniform task sets in the time-triggered environment, except we set a upper bound of 3600 instead of 6000. However, after generating the time-triggered tasks, we change the tasks to a event-triggered task. We use the period and computational time generated by the log uniform task set to generate a new arrival task. We will have two different distributions between the arrivals of the event-triggered task. We will add a random release jitter on top of the period. This release jitter will be in the range of $[0, S]$, where S is a maximum factor of the release jitter, as shown in Equation 7.1.

$$A_i = T_i + (dist(0, S) * T_i) \tag{7.1}$$

Uniform distribution The first distribution is a uniform distribution of the added arrival time. We multiply the period with a random value between 0 and S (Equation 7.1), where everything between 0 and S has an equal chance. This distribution is visualized in Figure 7.8 (left). This distribution is chosen to have a truly random interval.

Half-normal distribution The same applies to the half-normal distribution with a maximum value S. We multiply the period with a random value between 0 and S and add this to the period for the next arrival. However, in this case there is a higher chance that lower values are chosen. The distribution is visualized in Figure 7.8 (right). This distribution is chosen to simulate tasks that encounter a delay, where smaller delays are more common.

Design of the experiment

We will evaluate the time-triggered policy based on the number of deadline misses in a runtime of 50 hyper periods. During the experiments we will change the following parameters:

- **Number of tasks**: We measure the effects on the deadline misses with respect to the number of tasks in the system. We cover the following number of tasks in our experiments: [1..10]. More than 10 tasks resulted in simulation times that were too long.

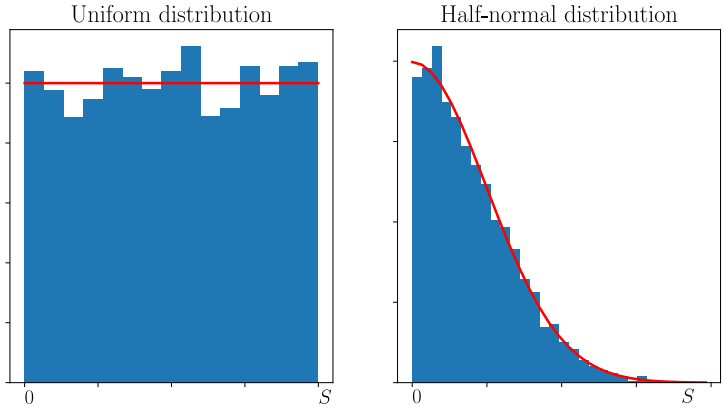


Figure 7.8: Types of jitter distributions

- **Utilization**: The effects of the utilization on the deadline misses are also measured with the following values: [0.1, ..., 0.9] with intermediate steps of 0.1.
- **History**: We pick different history values for our arrival curve which gives us more candidates for the lower bound values, discussed in Section 6.4. We pick the following history sizes: [1, 5, 10].
- **Release jitter factor S** Finally we vary the values S in our distribution, leading to a higher unpredictability as the values rises. We use the following values: [0.5, 1, 2, 4].

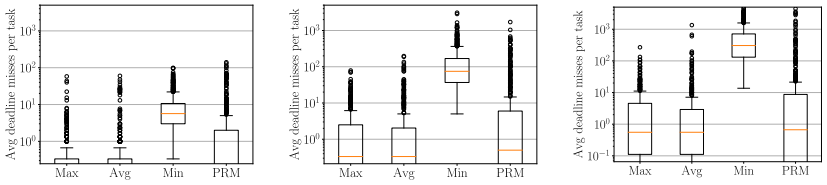
All experiments are performed with a maximum arrival time factor of 1 and use a uniform distribution, unless stated otherwise. In each graph we show 4 different policies: ArC-EDF where we use the minimum inter-arrival time (most left curve), the maximum inter-arrival time, the average inter-arrival time and Precautious-RM (PRM).

7.3.2 Results

For the results we have calculated the average deadline misses per task in the system. This means that we divide the total number of deadline misses by the amount of tasks in the system. The number of deadline misses are plotted on a logarithmic scale since the average and maximum amount of deadline misses are are very far apart.

Impact of the number of tasks on the number of deadline misses In this experiment we vary the number of tasks in the system and measure the number of deadline misses using a utilization of 0.8. The results are shown in Figure 7.9. We see that if we increase the number of tasks the number of deadline misses using the minimum inter-arrival time increases relatively to the other policies. The other policies using the average and maximum inter-arrival time and PRM do increase in deadline misses, but both increase equally. PRM

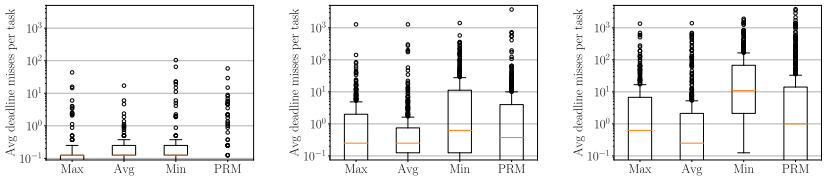
only shows some slight more variance in the number of deadline misses relative to the maximum and average inter-arrival times. The reason the minimum inter-arrival time performs bad (up to 2 orders of magnitude), is due to the increasing number of idle-time insertions where this was not needed since the actual arrival was a lot later. So the system inserted an idle-time while this was not needed, increasing the amount of deadline misses.



(a) $n = 3$ (b) $n = 6$ (c) $n = 9$

Figure 7.9: Impact of the number of tasks on the number of deadline misses

Impact of the utilization on the number of deadline misses Varying the utilization in the system has a slight lower effect on the numbers of deadline misses. For this experiment we used 8 tasks. Looking at Figure 7.10 we see that the minimum inter-arrival time is in all cases the worst option. While increasing this utilization we see all policies have an increasing number of deadline misses. Again, the reason for the performance of the minimum inter-arrival time is the insertion of too many idle-times.

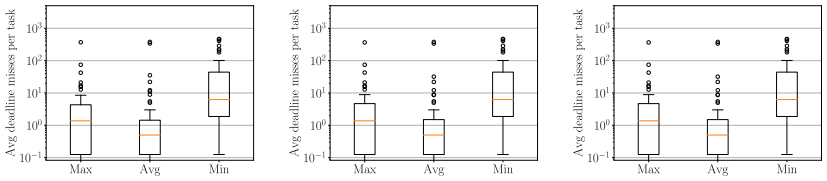


(a) $U = 0.3$ (b) $U = 0.6$ (c) $U = 0.9$

Figure 7.10: Impact of the utilization on the number of deadline misses

Impact of the history size on the number of deadline misses In this experiment we evaluation the effect of the history. We set the values of $n = 8$ and $U = 0.8$. The results (Figure 7.11) are pretty clear: the history size has *no* effect on the number of deadline misses. Therefore we didn't measure the overhead of ArC-EDF, since we were interested how the history size would effect the overhead compared to the deadline misses. If there is no improvement with using more history, the best option is a history of 1. Hence, we can conclude that using an arrival curve has no benefits for random event-driven systems.

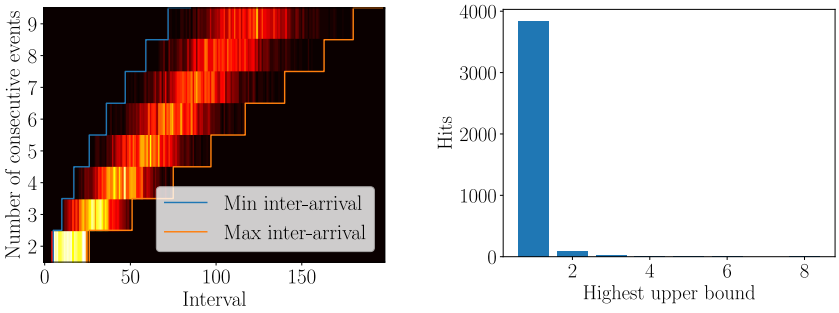
Reasons why there is no impact Because of the low impact of the history, we explored why this was the case. We gained some insights that using



(a) M = 1 (b) M = 5 (c) M = 10

Figure 7.11: Impact of the history size on the number of deadline misses

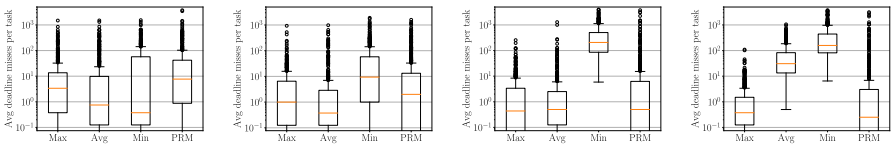
more history only adds less likely scenario's to the table. In Figure 7.12a we see an arrival curve with a uniform distributed release jitter. Here we see that the inter-arrival time is indeed uniform distribution between two consecutive events. But if we increase the consecutive number of events, we notice that the minimum inter-arrival curve starts falling behind on the more common inter-arrival values. This is because it is less likely in a uniform distribution to have the lowest value multiple times in a row. So ArC-EDF is only taking less likely occurrences into account when increasing the history. We also see this in 7.12b, where we plotted the highest lower bound when estimating the next arrival. Note that we used a history size of 15 for this figure. Over 96% the inter-arrival time between two consecutive events is the highest lower bound and is used for the next arrival. History size over 9 was not selected even once. For this test we ran 500 task sets with $n = 6$, $U = 0.6$ and $S = 2$.



(a) Distribution within arrival curve (b) Highest upper bound occurrences

Impact of the maximum arrival time factor on the number of deadline misses In this final experiment we look into the effects of the value S on the number of deadline misses. Looking at Figure 7.13 we see very different results for different factors. If we use a low factor, where tasks are encountering low jitter and are relatively close to behave as periodic tasks, the minimum inter-arrival time is the best policy (which is the same as CW-EDF). This corresponds with the fact that CW-EDF has a higher schedulability for periodic tasks. If we increase the factor, where the arrivals of the tasks become more unreliable we see that the minimum inter-arrival time starts for perform bad. This is again due to the fact that the minimum inter-arrival time will insert too many idle-times since it will insert an idle-time if one of all other non-pending tasks will

miss their deadline. However, this deadline is too pessimistic since all tasks will arrive later. Also interesting is that PRM performs quite good compared to ArC-EDF. This is because PRM will only insert an idle-time when the highest priority task will miss the deadline, thus will insert less (incorrect) idle-times. It is also interesting to see that eventually the average inter-arrival time start to perform worse.



(a) S = 0.5 (b) S = 1 (c) S = 2 (d) S = 4

Figure 7.13: Impact of the maximum arrival time factor on the number of deadline misses

Comparison to work-conserving While this thesis focuses on non-work-conserving policies, we thought it would be interesting how work-conserving policies compare. We noticed in previous experiments that inserting too many idle-times did cause more deadline misses. In Figure 7.14 we see that the work-conserving policies NP-RM and NP-EDF (as discusses in Section 3.2) are performing as good as ArC-EDF with the maximum inter-arrival time and PRM. With these results we see that non-work-conserving policies are not preferable in terms of deadline misses and even are less preferable in terms of overhead compared to the work-conserving policies.

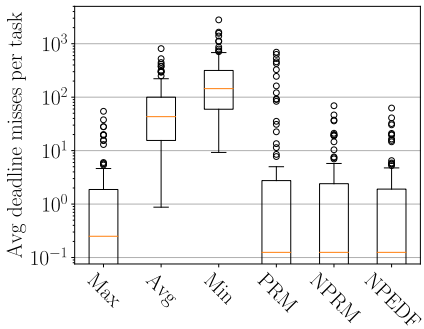
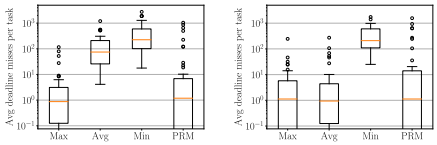


Figure 7.14: Comparing to work-conserving policies

Impact of the arrival time distribution on the number of deadline misses We look at the impact of the distribution of the arrival time on the amount of deadline misses. Values are set to $S = 2$, $U = 0.8$ and $n = 8$. Looking at 7.15 we see that having a half-normal distribution has a negative effect on the deadline misses for the average arrival time. This is because the average value is not the best way to represent an half-normal distribution, the median

might be a better representation for half-normal distributions. In both cases the maximum inter-arrival time and PRM have the roughly the same amount of deadline misses on average. The minimal inter-arrival time is again the worst option since this insert too many idle-times.



(a) Half-normal (b) Uniform

Figure 7.15: Impact of the arrival time distribution on the number of deadline misses

Chapter 8

Conclusions

In this chapter we present the conclusions of this thesis, where we introduced two new policies and will answer the two research questions.

8.1 Summary of contributions

Non-work-conserving policy with low overhead With KP-EDF we've created a non-work-conserving policy which has equal performance as CW-EDF, but with significantly lower overhead. Our policy is able to detect all critical tasks for the idle-time insertion policy and reduces overhead by ignoring the non-critical tasks. In the evaluation we see that we reduced the overhead up to 60%, while presumably this value increases with more tasks added to the system. The overhead of KP-EDF is $O(k \log k)$, where k is the size of the critical tasks which have an influence on the idle-time insertion decision. The overhead of CW-EDF is $O(n \log n)$ where n is the total amount of tasks in the system. Because the size of k relative to n is dependent on the type of task set, there is no exact reduction in overhead, but $k \leq n$ holds for every task set. We saw for log uniform task sets that k was at least 2 times smaller than n.

Event-driven policy that optimizes deadline misses ArC-EDF is an event-driven non-work-conserving policy which adapts to the inter-arrival time of event-triggered tasks. Using the arrival curve we can estimate the next arrival for the idle-time insertion policy. We have shown that using a history is not effective, so ArC-EDF drops down to having a single value as a period. The value selected as the period can be anywhere within the minimum inter-arrival time and the maximum inter-arrival time of the task. The evaluation has shown that the least amount of deadline misses were reached with the maximum or average inter-arrival time as the period, as the minimum inter-arrival time inserted too many idle-times which resulted in significant more deadline misses (up to 3 orders of magnitude).

8.2 Research questions

RQ1 *How can we lower the run-time overhead of CW-EDF but maintain its schedulability?*

We've managed to reduce the overhead by reducing the overhead of the idle-time insertion policy of CW-EDF. Since we keep reducing the overhead until we hit a different idle-time insertion decision, we guarantee that our policy has the same schedulability as CW-EDF. Reducing the overhead was reached by identifying the critical tasks of the idle-time insertion policy (IIP) and only evaluating these tasks in the IIP. In the evaluation we see that the number of critical tasks is very low compared to the total number of tasks in the system. This leads to the reduction of overhead of 60% in task sets with 12 tasks, where it's implicated that relatively more overhead is reduced by having more tasks in the system.

RQ2 *How can we add support for event-triggered tasks in a non-work-conserving policy?*
We added support for event-triggered tasks by analyzing and storing the behavior of the event-triggered task. While existing non-work-conserving policies will use the minimum inter-arrival time of the task, our solution has shown that it is in favor of the number of deadline misses to use another value (between the minimum and maximum inter-arrival time). We also see that having a history shows no gain in terms of deadline misses, so overhead remained as low as CW-EDF.

8.3 Future work

In this thesis we've investigated non-work-conserving policies. We did rule out the usage of an arrival curve. There is however still some future work yet to be explored:

Critical tasks analysis We found that having large period ratios result in a lower critical task set size. Being able to calculate this set instead of performing CW-EDF as an offline preprocessing tool might be more efficient as we don't have to run CW-EDF until we reach the hyperperiod. If this analysis has very low overhead, it might even be interesting to calculate the critical tasks online. The evaluation of the critical task set size of automotive task sets (Section 7.2) indicates that a period ratio of higher than 2 has a great impact on being critical or non-critical, but this is not proven.

Critical event-triggered tasks Finding the critical set for event-triggered tasks was hard as each idle-time decision is influenced by the arrival of the next job, which is unknown for event-triggered tasks and the critical tasks can be different for each hyperperiod run. But it would be interesting to see which tasks are always in this set, how this is different from time-triggered tasks and if this holds for task sets with a combination of time and event-triggered tasks.

History size While we've detected that history size has no benefits with respect to deadline misses, this is possibly only the case for truly random release jitter. If a task has different behaviors, where the release jitter is high but constant and drops at a later point in time, the arrival curve and history might have an impact. Also we saw effect of the distribution on the amount of deadline misses. There might be tasks with an arrival time distribution and behavior that do benefit from using a history.

Deadline misses analysis We've shown that using the arrival curve is only useful for a history of 1, and that in most cases the maximum inter-arrival time gives the least number of deadline misses. However, the evaluation is not consistent in what the best inter-arrival time is. The more unpredictability we have in our system, meaning that the smallest inter-arrival time is a lot different than the average inter-arrival time, the better the maximum inter-arrival time performs. However in a more predictable environment, where the average inter-arrival time is close tot the smallest inter-arrival time, the average or even the minimum inter-arrival time performs the best. The exact crossing point of these lines is yet to be determined. We also saw that the distribution has a great impact on using the average inter-arrival time. For tasks with no uniform distribution of the release jitter, it is likely that the minimum arrival time line results in the lowest number of deadline misses.

Bibliography

[1] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis. A survey of industry practice in real-time systems. In *2020 41st IEEE Real-Time Systems Symposiums (RTSS)*, 2020.

[2] G. Buttazzo. *Hard Real-Time Computing Systems*. 24. Springer US, 3 edition, 2011.

[3] G. Carvajal, M. Salem, N. Benann, and S. Fischmeister. Enabling rapid construction of arrival curves from execution traces. *IEEE Design & Test*, PP, 2017.

[4] P. Emberson, R. Stafford, and R.I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 6–11, July 2010.

[5] L. George. Preemptive and non-preemptive real-time uni-processor scheduling, 1996.

[6] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95.

[7] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of period and sporadic tasks. In *[1991] Proceedings Twelfth Real-Time Systems Symposium*, pages 129–139, 1991.

[8] S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[9] M. Nasri and B. B. Brandenburg. Offline equivalence: A non-preemptive scheduling technique for resource-constrained embedded real-time systems (outstanding paper). In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 75–86, 2017.

[10] M. Nasri and G. Fohler. Non-work-conserving non-preemptive scheduling: Motivations, challenges, and potential solutions. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 165–175, 2016.

[11] M. Nasri and M. Kargahi. Precautious-rm: A predictable non-preemptive scheduling algorithm for harmonic tasks. *Real-Time Syst.*, 50(4):548–584, July 2014.

[12] T. Pop, P. Eles, and Z. Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)*, pages 187–192, 2002.

[13] B. Sanjoy and B. Enrico. Partitioned scheduling of sporadic task systems: an ilp-based approach. 2008.

[14] T. Shepard and J. A. M. Gagne. A pre-run-time scheduling algorithm for hard real-time systems. *IEEE Transactions on Software Engineering*, 17(7):669–677, 1991.

[15] P. Tabuada. Event-triggered real-time scheduling of stabilizing control tasks. *IEEE Transactions on Automatic Control*, 52(9):1680–1685, 2007.

Notations

Symbol	Description
t	Current time in the system
τ_i	Task i
T_i	Period of task τ_i
C_i	Computational time of task τ_i
D_i	Deadline of task τ_i
U_i	Utilization task τ_i
$J_{i,j}$	jth instance of task τ_i
H	Hyperperiod of a task set
r_i	Release time of task τ_i
r_i^j	Estimated release time of task τ_i with $j+1$ consecutive events
d_i	Absolute deadline of a job
A_i	Arrival curve of task τ_i
A_i^{j-max}	The maximum inter-arrival time of j consecutive events of task A_i
A_i^{j-min}	The minimum inter-arrival time of j consecutive events of task A_i
M	History size of ArC-EDF and height of the arrival curve

Glossary

arrival time The time instant at which a job enters the ready queue.

arrival curve Method to model the activation pattern of an event-triggered task.

computational time The amount of time required by the processor to execute a job.

deadline The time within a job should complete its execution.

dispatch Assignment of a task to the processor.

event-triggered Tasks where the inter-arrival time is not regularly interleaved.

hard real-time Jobs must be guaranteed to complete within their deadlines.

hyperperiod The minimum time it takes for the system to repeat itself.

inter-arrival time The time interval between the activation of two consecutive instances of the same task.

job A single instance of a task.

non-work-conserving A policy which can leave the processor idle while there are tasks in the ready queue.

offline Offline policies are policies that run before the execution of the system.

online Online policies are policies that run during execution of the system.

overhead The time required required by the scheduling policy.

pending task A task in the ready queue.

period The interval between two consecutive jobs of a periodic task.

policy An algorithm which decides the highest priority job to dispatch.

preemption A form of scheduling where jobs can be interrupted to execute higher priority jobs.

ready queue A set of jobs waiting for execution by the processor.

schedulability The ability of a policy to find a schedule without deadline misses.

soft real-time Jobs should complete before the deadline, but no there are no consequences if a deadline is missed.

tardiness The time a job finishes execution after the deadline.

task A functionality of the system.

time-triggered A task that is released at a constant rate.

utilization The fraction of the processor time utilizes by the computation of the task.

[0]Some of these definitions are from G. Buttazzo [2]

APPENDIX

Automotive Runnables Table

Period (ms)	BCET (us)	Avg ET (us)	WCET (us)
1	0,34	5,00	30,11
2	0,32	4,20	40,69
5	0,36	11,04	83,38
10	0,21	10,09	309,87
20	0,25	8,74	291,42
50	0,29	17,56	92,98
100	0,21	10,53	420,43
200	0,22	2,56	21,95
1000	0,37	0,43	0,46

Table 8.1: Runnable execution times