Delft University of Technology

Real Time Systems
CESE4025

Assignment A: Coding Schedulers

Authors:
Miguel Prazeres (6079067)

João Aragonez (6079059)

December 9, 2023

I. Understanding the System

Question 1

The *example* scheduler is not a real-time scheduler, since it does not take into account the constraints of the system, mainly the computation time and deadlines of each task. Instead of scheduling the tasks with a scheduling policy that ensures the execution of tasks in the correct time slots, it just rotates sequentially which task executes next, without any preemption or regard to deadlines or periods.

Question 2

The hyper-period for a set of periodic tasks is the least common multiple of the periods of the tasks. For the given task set, the hyper-period is $\text{lcm}(20ms, 40ms, 60ms) = 120ms$.

Question 3

The processor utilization factor for a task set of size **n** is given by the following formula:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

The chosen task sets all have the same hyperperiod (120ms) and are defined as follows (where Ci is the execution time and Ti is the period of the respective tasks):

T0	**C**	**T**
t0	2	20
t1	10	40
t2	30	60

T1	**C**	**T**
t0	5	20
t1	10	40
t2	30	60

T2	**C**	**T**
t0	5	20
t1	20	40
t2	24	60

The processor utilization factor for all the task sets:

$$\textbf{T0} : U = \frac{2}{20} + \frac{10}{40} + \frac{30}{60} = 0.1 + 0.25 + 0.5 = 0.85$$

$$\textbf{T1} : U = \frac{5}{20} + \frac{10}{40} + \frac{30}{60} = 0.25 + 0.25 + 0.5 = 1$$

$$\textbf{T2} : U = \frac{5}{20} + \frac{20}{40} + \frac{24}{60} = 0.25 + 0.5 + 0.4 = 1.15$$

II. Implementing a Rate Monotonic Scheduler

Question 4

Our implementation of Rate Monotonic consists in applying the algorithm assuming that the relative deadline is equal to the task period. Moreover, instead of calculating the priority as being the inverse of the period of a task, we just use the period of a task directly. Having this in mind, we added one field in the Task struct: *number_of_executions*. This field works as a counter that keeps track of how many times a task has already executed. It is mainly used for the preemption of the tasks with higher priority, as well as assuring that the tasks only execute once per period.

Our algorithm executes the following steps:

1. In the first iteration it orders the tasks by their period, from the task with the lowest period to the highest, since the tasks are periodic and their priorities are fixed.
2. Then, every iteration, it calculates the amount of time the scheduler should sleep. It is done by finding out what task, with higher priority relative to the selected one, has the closest release time relative to the current time.
3. Finally, if the previously active task comes with the **finished** value as true we increment its **number_of_executions** and then activate the task with the lowest period (highest priority) that hasn't run the number of times that it should until the current instant. If the previous task comes with the value as false, we just activate the new task without incrementing the **number_of_executions** of the previous task. I
4. If all the tasks already executed in the proper time period, the scheduler is set to idle mode.

As we can see in figure 1,which shows the executions of RM with the given task set, our implementation works properly, since: t0 executes immediately after its jobs are released; t1 executes as soon as t0 finishes; t2 executes only when the higher priority tasks are not executing; and the system is only idle when no task need to be executed.

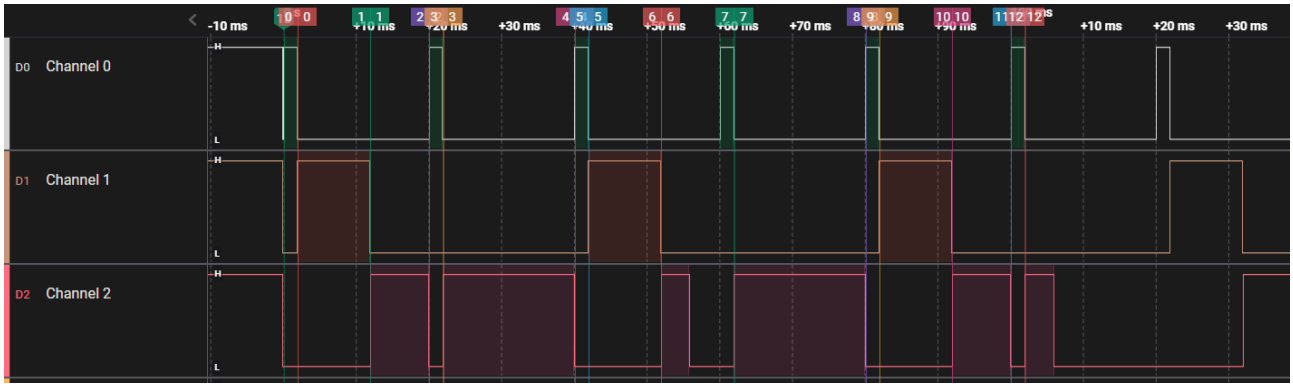

Figure 1: Execution of RM with given task set (U<1)

Figure 2 has the computation times of all tasks in one hyperperiod, when RM is applied. These represent the green, orange and purple fills, in cronological order, in Figure 1.

Question 5

The overhead of our RM scheduler corresponds to the time the scheduler needs to stop the current task and switch to the next task. Therefore, the total overhead, in a hyperperiod, is the sum of all the time taken when switching tasks.

The total overhead of RM execution of the task set provided is 0.444293ms, derived from figure 7, while the hyperperiod is 120ms. Therefore, the percentage of the hyperperiod spent on overhead is:

$$\frac{overhead}{hyperperiod} = \frac{0.444293ms}{120ms} = 0.0037ms = 0.37\%$$

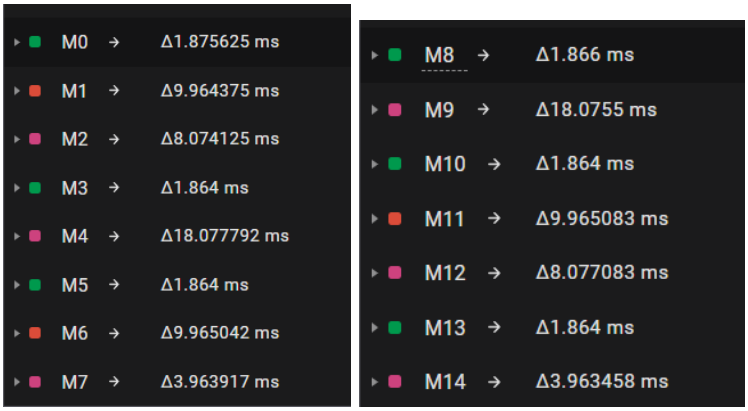

Figure 2: Measurements of the computation times of RM with task set U<1

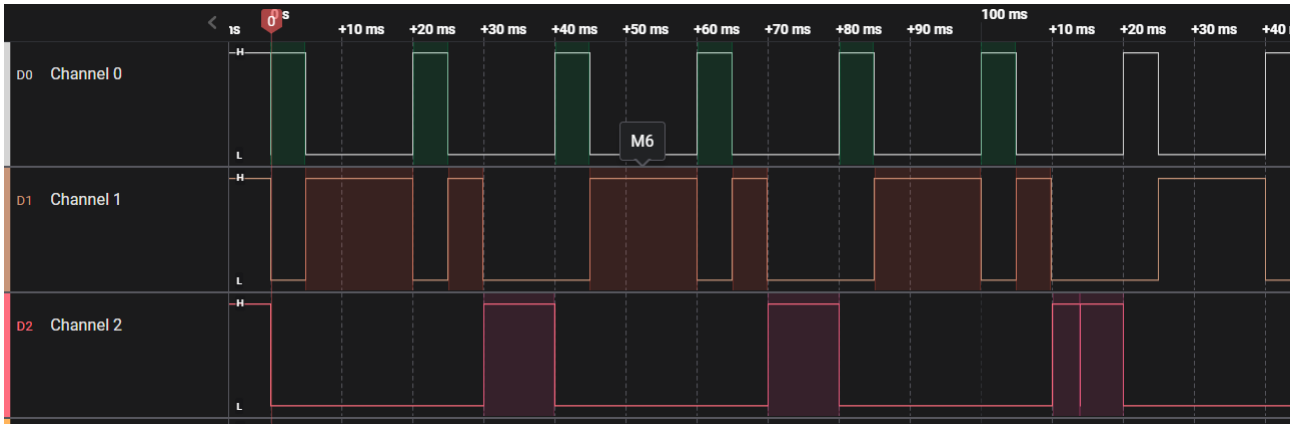

Figure 3: Execution of RM with task set with U>1)

Question 6

Overloading occurs when the task set is not schedulable and the tasks fail their deadlines, which means that the overall response time of the system is increased. Our scheduler deals with the overloading by, essentially, starving the tasks with lower priority, when needed. Since the tasks with lower priority are always preempted by the tasks with higher priority when they arrive, the lower priority tasks may never execute, if the computation time of the high priority tasks is big enough.

In the task set chosen with U<1, everything runs as expected: all tasks are executed when they should, which means the task set is schedulable.

On the other hand, the task set with U=1 should not be schedulable, because of the overhead of the scheduler. However, since the real computation time of the tasks is smaller than the theoretical, due to the hardware and source code imprecision, the U=1 task set behaves as the U<1, meaning it is schedulable.

Regarding the U>1, we can observe that the set of tasks is indeed unschedulable, since t2 keeps getting preempted by the two tasks with higher priority. In this specific task set, t0 and t1 are always executed in the right way, meaning the deadlines are respected, where t2 keeps being interrupted and being delayed.

III. Implementing an Earliest Deadline First Scheduler

Question 7

Our implementation of Earliest Deadline First consists in applying the algorithm assuming that the relative deadline is equal to the task period and that the task priority equals to the sum of the negated periods. Here is the formula:

$$P = \sum_{i=1}^{n+1} -T_i \quad , n = number\_of\_executions$$

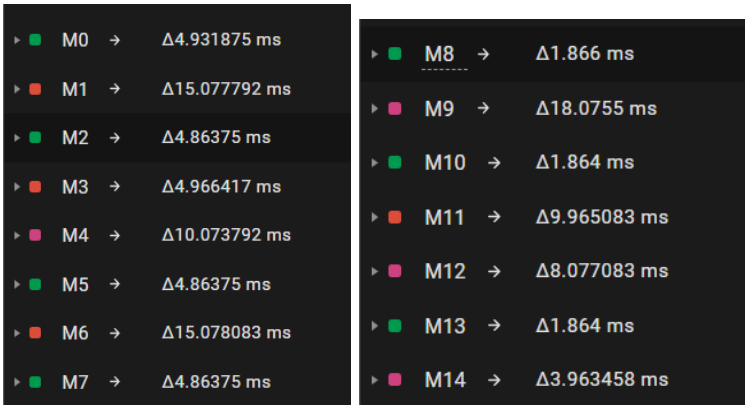

Figure 4: Measurements of the computation times of RM with task set U>1

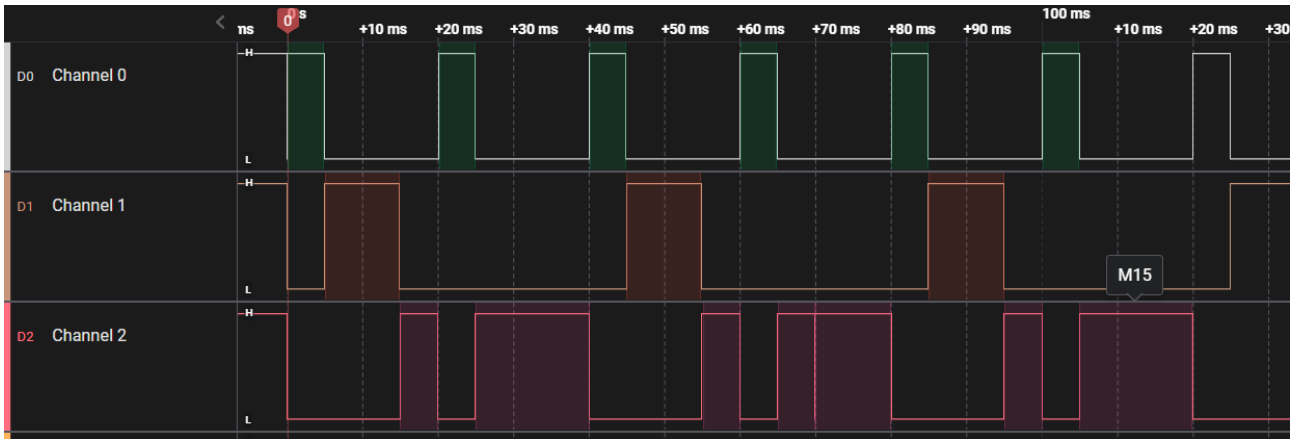

Figure 5: Execution of RM with task set with U=1

Therefore, we only added one more field to the Task struct: *priority*. We chose to calculate the priority this way, and not as the inverse of the period, to avoid floating points imprecisions in the calculations.

Our algorithm executes the following steps:

1. Firstly, it updates the number of completed executions and priority of the previous task, if it was already finished.
2. Then, every iteration, it selects the task with the highest priority that hasn't executed in its current period.
3. Finally, it calculates the sleep time of the scheduler, which is the nearest release time of the tasks with higher priority than the current task and sets active the chosen task. If all the tasks already executed in the current period slot, the nearest release time among all tasks is chosen and the scheduler is put on idle mode.

Question 8

Unlike RM, our implementation of EDF does not starve tasks with higher periods. Since the priority of a task is only updated when it is finished, all tasks are executed, since the priority is variable. Contrarily to RM, our implementation of EDF lets t2 finish, by delaying t0 and t1. This means that when t2 finishes, several jobs of t0 will execute in a sequence, since they have the highest priority.

Similarly to RM, our implementation of EDF works as expected with the task sets with U<1 and U=1 for the same reasons.

EDF can be better in task sets where the higher priority tasks have a significant execution time, since it does not starve the lower priority tasks. However, it delays all tasks, making the performance of the system overall slower (domino effect).

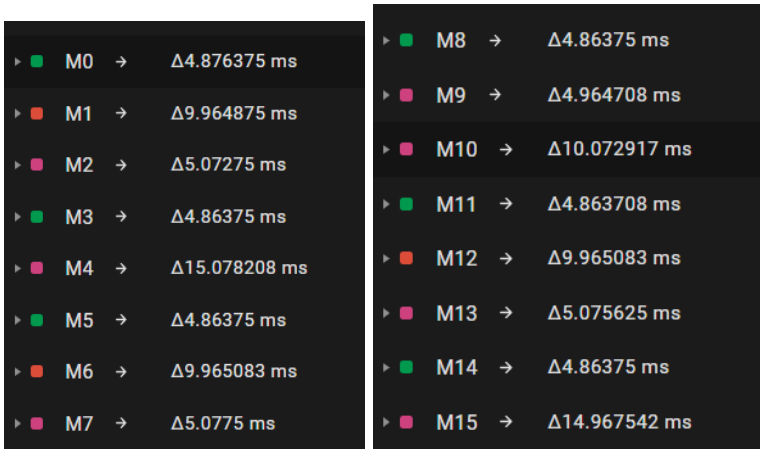

Figure 6: Measurements of the computation times of RM with task set U=1

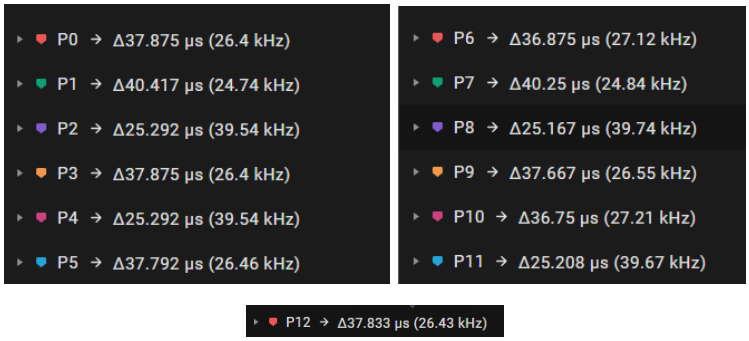

Figure 7: Measurements of the overhead with RM

RM on the other hand may be better if the higher priority tasks have lower computation values, since, although they always preempt the lower priority ones, the fact that their computation time is low means that there may be time for the lower priority tasks to execute.

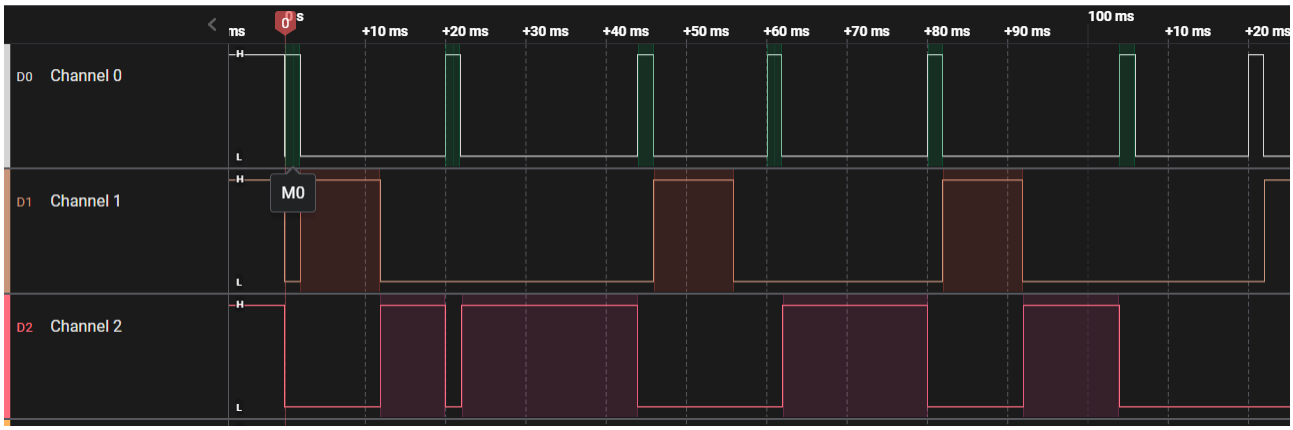

Figure 8: Execution of EDF with task set with U $<$ 1

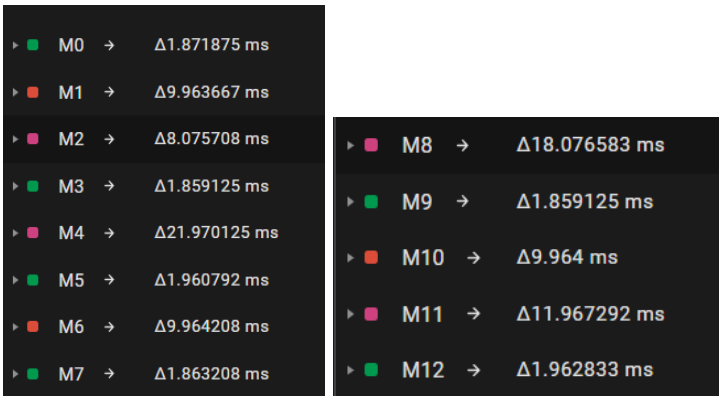

Figure 9: Measurements of the computation times of EDF with U $<$ 1

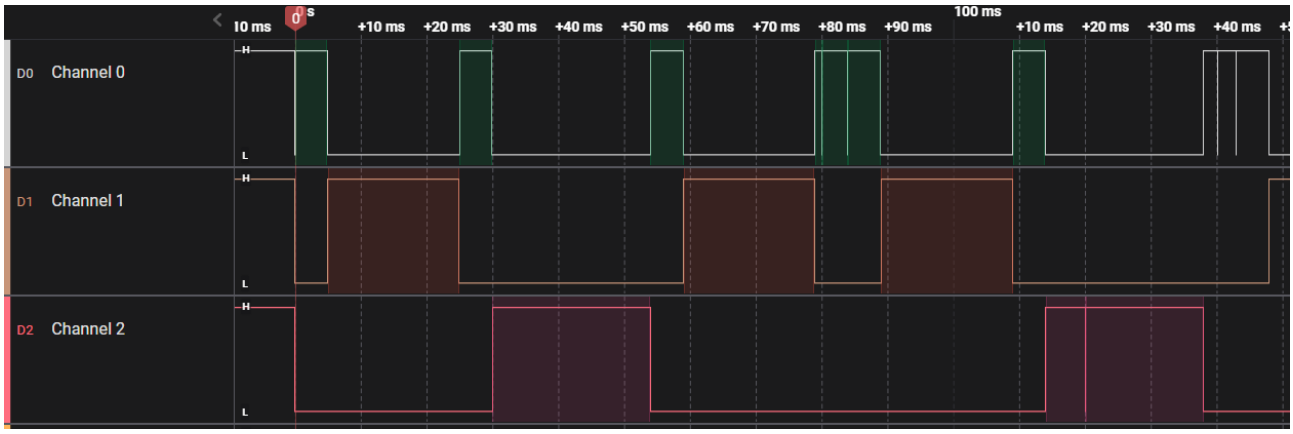

Figure 10: Execution of EDF with task set with U $>$ 1

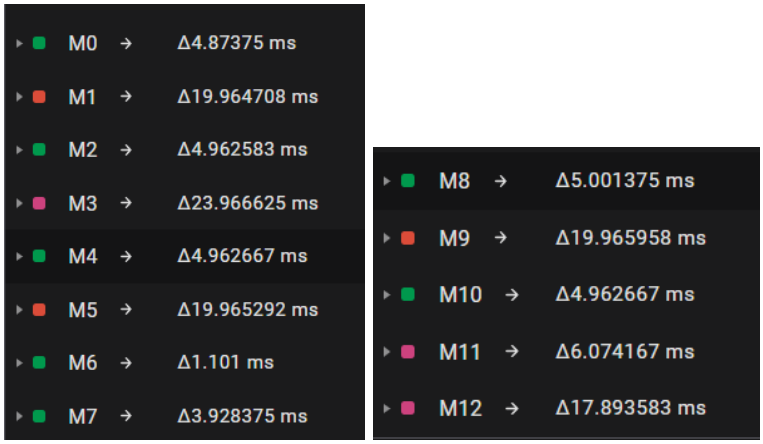

Figure 11: Measurements of the computation times of EDF with U $>$ 1

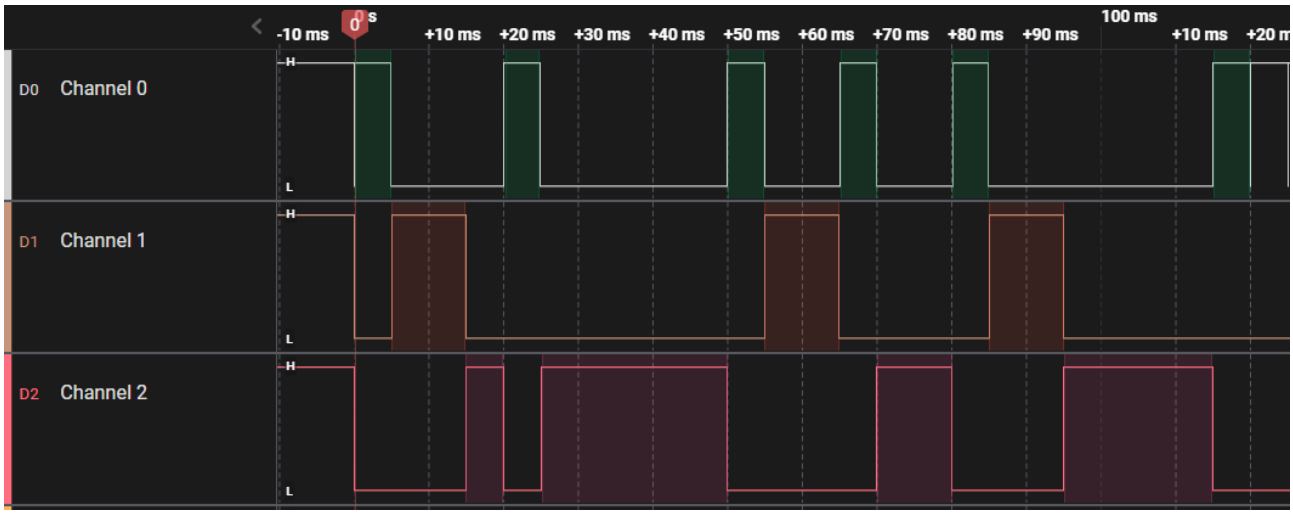

Figure 12: Execution of EDF with task set with U = 1

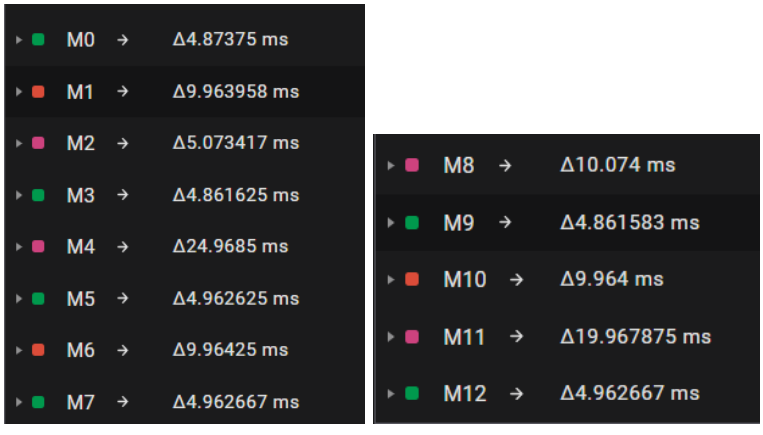

Figure 13: Measurements of the computation times of EDF with U = 1