Optimization of timeline scheduling for Time-triggered and Event-triggered tasks with Polling servers

02229 System Optimization

Alexander Barholm-Hansen	*s174624*
Henrik C. Nielsen	*s174651*
João Mena	*s223186*
Tiago Silvério	*s222963*
Tomás Estácio	*s223187*

Abstract

This report describes the project's design and implementations for the proposed optimization challenge in the course 02229 E22 Systems Optimization.

Abbreviations

List of abbreviations used throughout the report.

DM	Decision Maker
SoC	System-on-Chip
TT	Time-triggered
ET	Event-triggered
mt	Micro tick
EDF	Earliest Deadline First
EDP	Explicit Deadline Periodic
SA	Simulated Annealing
WCET	Worst-Case Execution Time
WCRT	Worst-Case Response Time

Contents

Part I

Project setup

1 Introduction

The foundation for this optimization challenge is to determine an optimal scheduling policy for a series of tasks on a single computation element, e.g., a core in a multi-core SoC. The case is based on the Advanced Driver Assistance Systems of modern vehicles, which are able to integrate a large number of complex functions such as lane changing, cruise control, or autonomous driving. As these functions are software rather than hardware based, and require some temporal and spatial isolation, scheduling of these tasks to ensure sufficient real-time capabilities is vital.

The combinatorial optimization problem which we aim to solve is simplified to an application model consisting of two sets of tasks: Time-triggered (TT) and Event-triggered (ET). TT tasks are periodic tasks which repeat the exact same activity (also called a job) in and infinite sequence. They are regularly activated with a constant period. ET tasks are sporadic and have a minimum period between which they arrive.

To schedule these tasks two approaches are used. TT tasks are handled with timeline scheduling based on the Earliest Deadline First (EDF) principle (see section 5). ET tasks are handled with polling servers which are TT tasks themselves and thus scheduled with these. Schedulability of ET tasks is analysed by the Explicit Deadline Periodic (EDP) principle (see section 6).

2 Solution requirements

The solution must ensure two primary constraints, (1) that both the TT and ET tasks are schedulable and complete before their deadlines. (2) The task separation constraints are satisfied, meaning that ET tasks with different separation values are handled by different polling servers. The optimization objective then becomes the average worst-case response times (WCRT) of all tasks, both ET and TT. The product of this challenge is thus an algorithm that produces an optimised solution consisting of the following whilst minimising the WCRT of all tasks:

Number of polling servers The number of individual polling serves we create which then become additional TT tasks.

Server properties For each polling server (task instance) we determine the period, budget, and deadline of the server. It should be possible to find a schedule for the TT tasks so they become schedulable.

Task to servers assignment To which server a sub-set of ET tasks are assigned to and handled by within the list of existing polling servers

3 Initial considerations

Below are described the initial assumptions and knowledge about the system.

3.1 Data sets and Application model

A number of data sets are given for the case on which to develop and test our algorithms. From these data sets we are given the following information. For each periodic TT task we know its period, worst-case execution time (WCET) and deadline. For each ET task we know the minimum inter-arrival time, WCET and deadline, as well as its priority. Additionally for the ET tasks, a separation value is given that determines which tasks are allowed to be assigned to the same polling sever. Tasks with a separation value of 0 are allowed to be assigned to any polling server. In table 1 an example of the test case data is shown.

Task name	WCET	Period	Type	Priority	Deadline	Separation
tTT0	55	3000	TT	7	3000	0
tTT1	13	4000	TT	7	4000	0
tTT2	45	3000	TT	7	3000	0
tET1	25	3000	ET	5	1845	0
tET2	19	4000	ET	2	2197	2
tET3	9	4000	ET	1	2977	1

Table 1: Example of test case data

3.2 Assumptions

To simplify the problem we make the following assumptions for all implementations of our optimisation algorithm:

- All tasks are considered preemptable, i.e., the schedule can be constructed such that tasks can be stopped to allow another task to run, and resumed afterwards.
- For a given polling server, the search space for the period can be between 2 and the hyperperiod of all TT tasks.
- We assume that a polling server's deadline is equal to the server period in all cases, thereby reducing the search space.
- The algorithm is allowed to visit solutions which are not feasible, either because the TT or ET tasks are not schedulable, during it's search. Keeping these solutions but adding a penalty value can improve the search for an optimal solution.

3.3 Experimental setup

Computer environment
The list below is showing the exact computer environment used throughout the project and results. The setup for the program e.g., the initial setup parameters within the program is listed under section 9 - *Results*.

- Model: Macbook Pro 13-inch 2019
- Maker: Apple inc.
- Operating system: macOS Ventura 13.0.1
- Number of processing cores: 4
- Types of processing cores: Efficiency
- Speed of processing cores: 1,4 Ghz
- Programming languages: Python v.3.10.5 64-bit
- Compiler: PyCharm 2022.2.3

Part II

Optimization algorithm

Our optimization algorithm uses the simulated annealing technique to determine an optimal solution. We have chosen SA since it is a conceptually simple algorithm to implement and therefore very versatile. Furthermore, SA implementations tend to be fairly robust [4]. This part explains each step of our implementation, from a general overview down to smaller details in vital functions. Section 5 describes the theory and implementation of scheduling TT taks. Section 6 details the scheduling and separation of ET tasks. Thereafter, section 7 describes SA and our implementation in detail. Finally, section 8 shows a working example of our algorithm.

4 Algorithm overview

This section gives a top-down overview of all the moving parts that make up the algorithm for our optimization strategy. The parts are self contained functions definitions in python that serve a specific function and are used as sub-routines in the main algorithm. On figure 1 below, a flowchart depicting the implementation of the entire algorithm is shown. The diagram is read from the upper left to the right. Coloured titles indicate where a specific and vital function definition is used. The entire chart is contained within the **main** definition.

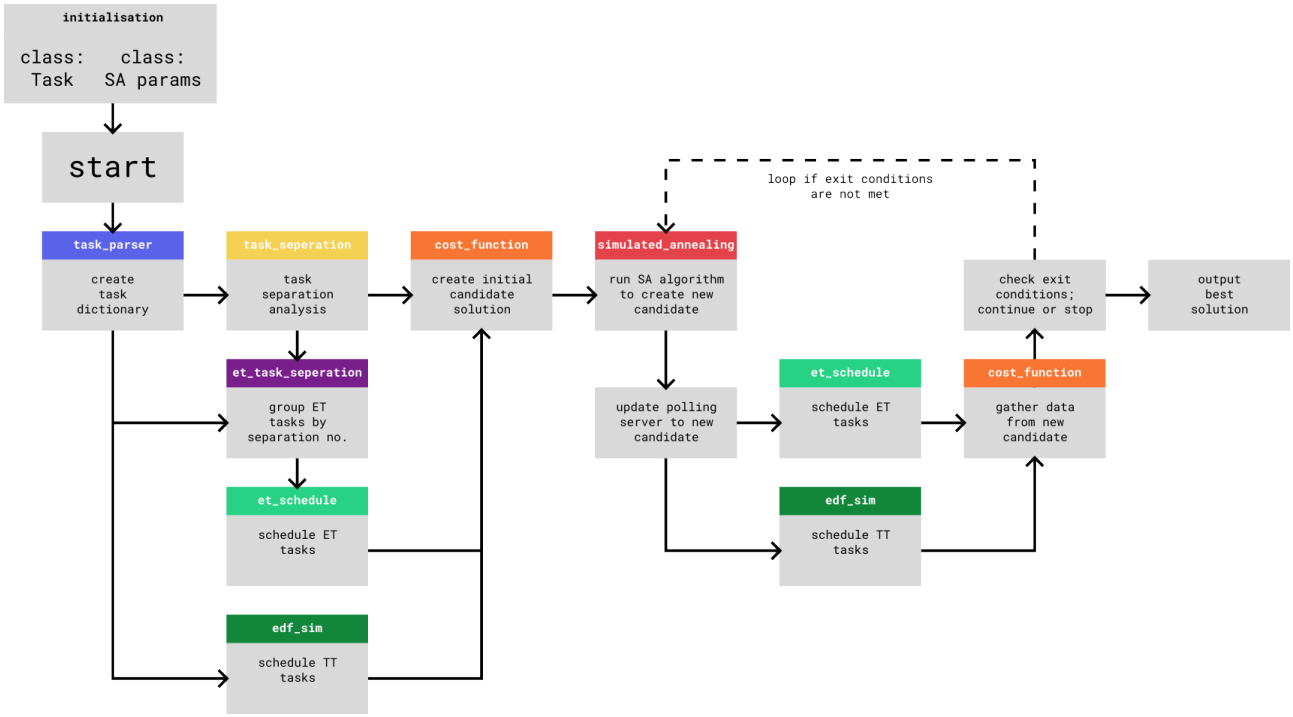


Figure 1: Diagrammatic representation of the optimization algorithm

Below is a short description of what purpose each part serves, as well as it's inputs and outputs, for all the parts of the algorithm as shown on the diagram above. A short pseudo-code is included for relevant parts. All algorithm parts are included as an appendix.

4.1 Primary functions

main - This is the primary part of the algorithm in which other parts are called, as illustrated in the diagram above. However, outside the **main** function, there are some additional parts that should be mentioned. These are:

Global variables used for accessing data in the task list containing all task information are defined. These are equal to the columns in the test case data sets; name, duration, period, type, priority, deadline and separation. It also defines global parameters used for the simulated annealing function. These are temperature, cooling factor, the best solution, the best solution cost, the best solution schedule, the current iteration and the maximum cost value to normalise candidate solution costs from.

Python libraries in addition to core features are loaded here. These include *pandas* for reading test case files in the .csv format, *Numpy* for handling two dimensional arrays and it's random number generator, *Random* for an additional random number generator, *Math* for additional mathematical operations (such as rounding numbers), *Logging* to use as a logging system for exporting data from an optimization run, *OS* for operations of reading and saving files to various paths and, finally *time* for measuring and exiting the simulation loop if a time condition is met.

task_parser - Reads the test case data sets containing the task properties from the specified path and file name. It then returns this information in a list and object format. This function uses the Pandas library for python to read .csv files. The inputs are *path* which specifies the location of the test case folder and *file* which specifies which test case file to parse. The output is *task_list* containing each task as an object with its properties accessible by the global Task parameters.

cost_function - Computes the cost of a candidate solution which is a combination of the worst case response time for TT and ET tasks. If the ET tasks are not schedulable, a penalty is applied to the computed cost. The inputs are *tt_wcrt* which is a list of the TT task WCRT, *et_wcrt_groups* which is a list of the ET task WCRT, and *et_sched* a Boolean for ET task schedulability. The output is *cost* which is an integer value describing the solution cost. See section 7 for a thorough description of this function.

simulated_annealing - Performs the simulated annealing optimization where a new candidate solution is generated from the neighbourhood of the current best solution. The inputs are *tt_tasks_wcrt* and *et_tasks_wcrt*, the WCRT for both TT and ET tasks. Then, *tt_schedule* is the schedule table for TT tasks and *et_tasks_sched* represents the number of polling servers in our solution that can't schedule the ET tasks assigned successfully. The *candidate_solution* which is the new solution to be evaluated and compared to the current best solution. Finally, the global *parameters*, the TT schedule *hyperperiod* and *tt_schedule_bool* if the TT tasks are schedulable or not. It returns *budget_poll_servers* and *period_poll_servers* which are all arrays

that define the properties of the polling servers for the new candidate solution. See section 7 for pseudo code and more details.

Scheduling functions

edf_sim - Schedules the TT tasks using the EDF principle and computes the worst case response time. The inputs are *task_list* containing all tasks and their properties in a dictionary format and *ps_array* which contains the information of the polling servers in the same structure as the tasks. It returns *sigma*, *wcrt* and *T*, which is an integer value describing the schedule hyperperiod. See section 5 for pseudo-code and a thorough description of this function.

edf - Returns the name of the task which has the earliest deadline and a duration that is different from 0, i.e that it has computation time remaining. As an input it takes *tt_tasks* which includes all the TT tasks and their parameters. It returns *name* which contains the name of the task with the earliest absolute deadline. See section 5 for pseudo-code and a thorough description of this function.

task_separation - Determines the minimum and maximum amount of polling servers to create for the given test case. Minimum amount is determined by the number of unique separation values larger than one present in the data set. Maximum amount is determined by the number of individual ET tasks. The input is the *task_list* containing all tasks and their properties for the given data set. The output is the maximum number of servers *max_no_ps* as an integer and the minimum number of servers *min_no_ps* as an integer. See section 6 for more details.

et_tasks_separation - Groups ET tasks by their separation value so that they can be separately scheduled by the polling server they are assigned to. As its input it takes *task_list* which is the dictionary containing all task information and *no_poll_srv* which is the current count of polling servers generated. It returns *et_tasks_all_groups* which is a modified array containing sub arrays similar to the *task_list* structure where all tasks with the same separation value are contained in a sub array. See section 6 for more details.

et_schedule - Determines if the ET tasks are schedulable given the polling servers they are assigned to. It also computes the WCRT for the ET tasks. As an input it takes *et_tasks* which is a list of ET tasks, *Cp* which is the task compute time, *Tp* which is the task minimum inter-arrival period and *Dp* the task deadline. It returns a Boolean value if the task set is schedulable or not as well as a tuple with the WCRT. See section 6 for a thorough description of this function.

priority_parser - Reassigns the priorities of the ET tasks for each polling server which a list of tasks is assigned according to the EDF principle. As its input it takes *et_tasks* which is a task dictionary containing only ET tasks. It returns the same dictionary with updated priority values for the tasks depending on their deadline. See section 6 for a thorough description of this function.

4.2 Utility functions

divisible_random - Generates a random integer from an interval between two supplied integers but only if the range is larger than a third integer. As its input it takes three integers; a, the start of the range, b, the end of the range and n, the integer which the range must be bigger than. Additionally it ensures that the generated integer is evenly divisible by n. It returns *result* the generated integer.

pdc - Calculates the Processor Demand Criterion for a list of TT tasks to determine if they are schedulable or not. It is also used to check if the created polling servers make the set of TT unschedulable. As input, it takes *task_list* which is the array of tasks to consider, *tt_hyperperiod*, the hyperperiod of the tasks considered, which is the time in which the schedule repeats itself, u, which is the U computed in *edf_sim*, *largest_deadline*, a value also determined in *edf_sim* that corresponds to the largest deadline from all the tasks in the set, and *deadline_list* which is a list with the value of all the deadlines from the task set considered. It returns 0 if it the task set is schedulable and 1 if it is not.

create_poll_srv - Creates a dictionary similar to the one the test case task set is stored in with the information of the created polling servers. This dictionary can be added to the task set dictionary so the polling servers can be scheduled along with TT tasks. As input, it takes two lists, *budget* and *periods*, of equal length that specify each servers budget and period. As mentioned in section 3.2 we assume server deadlines are equal to periods. It returns *ps_matrix* which contains the polling server name, budget, period, type, priority, deadline and separation.

supply_poll_server - Creates the schedule table of the polling server passed as argument. As input, it takes *tt_schedule* ,which is the schedule table of all tt tasks, and *poll_server*, which is a representative value of the polling server. This function returns a vector with the same size as *tt_schedule* and with value 1 at the instants which there is a job of the polling server studied executing.

5 EDF scheduling and Algorithm 1

5.1 Algorithm implementation

In several real-time systems, periodic tasks represent the majority of the computational demand of the applications. The algorithm applied in the project for scheduling the periodic tasks is an Earliest Deadline First (EDF) simulator, that takes as input time-triggered (TT) tasks parameters, such as the name of each task, the duration, the period, the type (in this case, all of them are TT), the priority (in this case, they all have priority 7) and the deadline. The output of the algorithm is the schedule table of the tasks and the worst-case response time (WCRT), which is an upper bound of the response times observed during the simulation.

In this project, we work with two different kind of tasks: time-triggered and event-triggered tasks. However, for this algorithm we will only analyze the first kind. A task is "a sequence of instructions that in the absence of other activities is continuously executed by the processor until completion"[3], and being a time-triggered task means that the instructions are initiated periodically at a specific time in the real-time system.

Scheduling can be described as allocating resources and all processes that require computational power in a way that all the time parameters are met and accomplished successfully. The EDF is a dynamic scheduling rule that selects tasks according to their absolute deadlines, meaning tasks with earlier deadlines will be executed first (with higher priority). The algorithm used in the project has constrained deadlines, because all the deadlines associated with each task are less than or equal to their periods. The created program is also a means of calculating the worst-case behavior for the response time of the tasks in the scheduling table obtained.

The first approach creating the algorithm 1 described in the project guide is to parse the information contained in the "test cases" folder provided, which contains different *task sets* with all the TT tasks parameters and we store these results in different arrays to use them throughout the algorithm. Since the polling servers created in a different program called *create_pol_srv* are also periodic, we parse that information also in the same array, treating them as normal TT tasks. Next, we determine the processor utilization factor (U), to calculate the fraction of time spent on actually executing tasks, by using the formula:

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

being n the number of TT tasks used in the program.

We also determine the hyperperiod of the tasks by calculating the least common multiple of periods for the TT tasks using the formula:

$$H = lcm(T_1, ..., T_n),$$

being n the number of TT tasks used in the program, since after the determined value, the schedule repeats itself, so we make it the stopping parameter for the creation of the schedule table. After knowing the hyperperiod, we go through each slot in the schedule table one micro tick (mt) at a time and check for each task:

- if its deadline has been reached and there's still computational time left, that means there has been a deadline miss and the schedule table returned is empty.
- if the task in question finishes his computation and the deadline has not been reached, its required to verify if the response time achieved is bigger than the previous maximum, storing the new maximum just once, for each time the task finishes. This value is saved for each task in a worst-case response time (WCRT) array, in order to return it after constructing the schedule table.
- if the period of the task has arrived again, there's the need to update the parameters associated with each task, such as: the computation time needs to be reset, the absolute deadline is computed, by adding the present simulation time and the deadline of the task in question, and the release time is stored.
- if there are still tasks with computation time. In the case there are still tasks with computational time left, we break the search loop and need to compute them. Otherwise, we assign that slot in the schedule table as an *idle* slot.
- if there are still tasks that have computation time, we perform the EDF algorithm to verify the order that the tasks need to be computed in: the tasks who have computational time left and have the smallest deadline are the first to be scheduled. Afterwards, we decrement one micro tick of the computational time.

Finally, after the simulation time reaches the value of the hyperperiod and the schedule table is created, we need to verify that all the tasks have a computational time equal to 0, or else the schedule table created is infeasible and should return empty. Also, to have a better understanding of the solution obtained from the algorithm, it is important to print out the schedule table obtained, along with the array of worst-case response time calculated.

In order to comprehend more clearly the main part of algorithm 1, which is the EDF algorithm, the following simplified code describes how it was implemented in our program.

```
""" Function that chooses the task with the earliest deadline,
among the tasks that still need to be executed (duration > 0). """

""" Set trade to the largest integer number, to guarantee that it is changed """

for Task in TT_Tasks_List:
    if trade is larger than Deadline_of_Task and Duration_of_Task is different from 0, then:
        trade ← Deadline_of_Task
        name ← Name_of_Task

return name, variable containing the name of the task with earliest deadline

""" Input format
tt_tasks = { 'task1': {'key_A': 'value_A'},'task2': {'key_B': 'value_B'}}

Output format
name = "tTT0" """
```

5.2 Implementation of Extension 1

To improve algorithm 1 we can check if there exists a solution to schedule the TT tasks before we spend the computation time running the EDF algorithm by using the Processor Demand Criterion (PDC) as described in [2] and assuming that all task deadlines are equal or less than the task period.

According to the author, a set of synchronous periodic tasks with relative deadlines less than or equal to periods can only be scheduled by EDF when:

$$\forall t > 0 \quad dbf(t) \leq t$$

It is, however, possible to verify that condition using a reduced interval of test points. Following the Theorem 4.6 in [2], if U < 1, the schedulability of a set of synchronous periodic tasks with relative deadlines less than or equal to the periods can be scheduled by EDF if

$$\forall t \in \mathcal{D} \quad dbf(t) \leq t$$

Where

$$\mathcal{D} = \{d_k \mid d_k \leq min[H, max(D_{max}, L^*)]\}$$

and

$$L^* = \frac{\sum_{i=1}^{n} (T_i - D_i)\, U_i}{1 - U}$$

The function $dbf(t)$ refers to the Demand Bound Function, which can be expressed by

$$dbf(t) = \sum_{i=1}^{n} \left[\frac{t + T_i - D_i}{T_i}\right] \cdot C_i$$

Below is shown the simplified code for the implementation of the *pdc* function in python, which computes the processor demand criterion by the theorem mentioned above. It starts by checking if the U passed as argument is less than 1. The computation of U happens in the *edf_sim function*. After that, it computes L^* and uses it to determine the list of testing points $\mathcal{D}$. Finally, for each instance of t in $\mathcal{D}$, it computes the demand bound function to check if $dbf(t)$ is greater than t, which would mean that the task set is not schedulable. Otherwise, it is schedulable.

```
### Simplified code of pdc function ###

"""
Input format
t_list = { 'task1': {'key_A': 'value_A'},'task2': {'key_B': 'value_B'}}
tt_hyperperiod = 12000
u = 0.88
largest_deadline = 4000
deadline_list = {3000 , 3998, 4000}

Output format
return = 0
"""

""" use processor demand criterion to determine if a task set is schedulable or not """

    if U > 1, then:
        return 1

    for Task in TT_Tasks_list:
        l_sum ← lsum + (Period_of_Task - Deadline_of_Task) * (Duration_of_Task / Period_of_Task)

    L* ← l_sum / (1 - U)

    check_condition = min(tt_hyperperiod, max(largest_deadline, L^{*))

    for Deadline in List_of_Deadlines:
        if Deadline <= check_condition:
            D[i] ← Deadline

    for t in D:
        for task in TT_Tasks_List:
            dbf ← dbf + ((t + Period_of_Task - Deadline_of_Task) / Period_of_Task) * Duration_of_Task
        if dbf > t, then:
            return 1

    return 0
```

6 EDP analysis and Algorithm 2

6.1 Algorithm Implementation

As we explained before, Event-triggered tasks are handled by polling servers. Each polling server is a TT task and a set of ET tasks is assigned to it. We have the necessity of analysing the ET tasks to know if they are schedulable for the polling server assigned to them. So we use the Explicit Deadline Periodic algorithm (EDP), to find if the ET tasks assigned to each polling server are schedulable and the WCRT of each ET task in that case.

Each ET task is characterized by its name, computation time , task priority, relative deadline time and minimal inter-arrival time, and separation. The separation factor is the one responsible to define groups of separations, tasks with different separation values can't be assigned to the same polling server. In section 6.3 we explain how to ensure that tasks with different separation values are not assigned to the same polling server.All the other parameters, except the name, are crucial to study the schedulabilty of ET tasks assigned to a polling server and also their WCRT. The period, computation time and relative deadline time of the polling server studied are also crucial for the ET schedulability analysis.

The main idea of the algorithm described in the project guide is to find the earliest instant which the linear supply bound function of the polling server and the maximum load function of each ET task intersect. That instant represents the WCRT of each ET task and finding this value we know the schedulability, if any of the ET tasks has a WCRT higher than its deadline, the ET tasks set is considered unschedulable. This analysis is done for all ET tasks set assigned to each polling server.

At first, we calculate Δ which represents the maximum time between the moment a job ends and the start of the next job of the polling server studied. In this first approach, we define Δ value as the longest possible time:

$$\Delta = T_p + D_p - 2 \cdot C_p$$

Then, we calculate the utilization factor of the polling server:

$$\alpha = \frac{C_p}{T_p} \leq 1$$

These two parameters are needed to calculate, for each instant in the hyperperiod, the linear supply bound function of the polling server studied.

After defining Δ and α, we study each task one at a time. At the beginning of the study we define the time to 0 and the task response time as the deadline value +1 because when response time is higher than the hyperperiod, the value is already assigned. After doing this for every instant, from 0 until the time reaches the hyperperiod, the linear supply bound function is calculated:

$$lslbf(t) = max\{0, (t - \Delta) \cdot \alpha\}$$

For each instant is also computed the maximum load of each task, evaluating the task demand:

$$H(t) = \sum_{\forall \tau_j \in \mathcal{T}^{ET}, p_j \geq p_i} \frac{t}{T_j} \cdot C_j$$

As we can notice following the maximum load formula, we can study the task demand only using the tasks with the same or higher priority than the actual task.

This way, calculating the maximum load of each task and the supply function of the polling server for each instant, we check in every instants if the value assigned to the supply function is higher than the maximum load . When that happens, it's considered that the WCRT was found and we need to check its value. If the WCRT found is lower than than the relative deadline time, we consider the current ET task is schedulable else it's not schedulable and the function returns False and the WCRT of the current task and also of the tasks studied previously.

6.2 Implementation of Extension 2

The test cases considered for this project contain the priorities of the ET tasks as inputs. However, since our solution for the project uses multiple polling server to handle different subsets of the ET tasks, it would be beneficial to reassign the priorities of those tasks to improve their schedulability. In order to do so, we believe that the best way to reassign the priorities would be to assign the biggest priority to the ET tasks with the lower duration, because, if the first ET tasks to be executed are the ones with lower duration, then it is more likely that the first tasks will exhibit a smaller worst-case response time. That way, with smaller worst-case response times, the cost of our schedule table will decrease and that's one of our main goals with this project. For a deeper understanding of the algorithm created for this extension, the following figure represents the simplified code for the algorithm:

```
"""
Reassignement of the priorities of the ET tasks according to lowest duration, biggest priority:
    1) check the biggest duration of the ET tasks in the test case
    2) divide that by the number of possible priorities in the test case
    3) assign priorities based on the space division
"""
Big_ct ← 0

""" Looks for the value of the largest duration in ET tasks set """
for Task in ET_Task_List:
    if Big_ct is smaller than Duration_of_Task, then:
        Big_ct ← Duration_of_Task

""" Gets the number with no decimal cases"""
```

```
chunksize ← Big_ct / 7

""" Rounds up the number if it has decimal cases """
if  Big_ct % 7 is larger than 0, then:
    chunksize ← chunksize + 1

""" Reassign the priorities based on chunksize and durations of ET tasks """
for Task in ET_Task_List:
    for i ← 0 until i == 6, incremeting 1 in each iteration:
        if Duration_of_Task is smaller or equal than chunksize*(i+1), then:
            Priority_of_Task ← 6-i
            break the loop

return ET_Task_List with priorities reassigned
```

6.3 Separation ET tasks

Due to the separation values constraint referred before, we implemented 2 functions to to solve it. These are *task_separation* and *et_tasks_separation* as described in section 4.

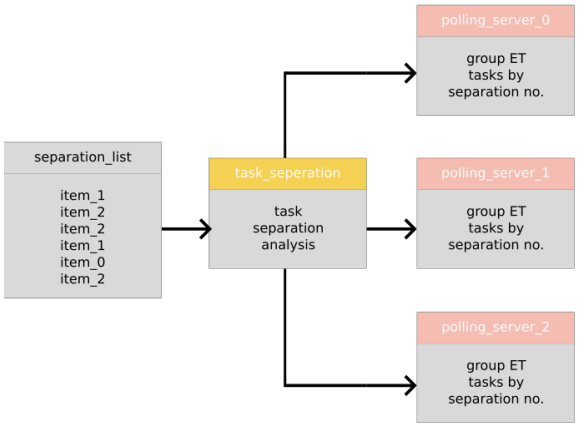


Figure 2: Separation ET tasks

task_separation is responsible for calculating the minimum number of polling servers needed to assign all the ET tasks of the data set used and also the maximum number of polling servers possible.

et_tasks_seperation receives as an argument the minimum number of polling servers needed, which will be the number of polling servers in our solution, and taking into account the separation value of each ET task, it separates them into different sets. Each set will be assigned to different polling servers. The ET tasks with separation value 0 are assigned taking into account the sum of all budgets of the ET tasks assigned to each polling server. This way we are ensuring that there are no polling servers with much more work than others . The diagram above

illustrates what these functions do. Below, the generic implementation of the two functions are shown:

```
### simplified code for task_separation ###
"""
Input format
task_list = { 'task1': {'key_A': 'value_A'},'task2': {'key_B': 'value_B'}}

Output format
max_no_ps = 23
min_no_ps = 3
"""
initialise task_seperation(task_list):
    sep_list ← [ ]
    count ← 0
    for task in task_list do:
        if task_type equals ET:
            append task_separation to sep_list
            count = count + 1
    unique_Values = set(sep_list)
    min_no_ps ← length(unique_values)
    max_no_ps ← count
    return max_no_ps, min_no_ps
```

```
### simplified code for et_tasks_separation ###
"""
Input format
task_list = { 'task1': {'key_A': 'value_A'},'task2': {'key_B': 'value_B'}}
no_polling_servers = 3

Output format
et_task_list = { 'group1': {'task1': {'key_A': 'value_A'},'task2':
{'key_B': 'value_B'}},'group2': ... }
"""
initialise et_tasks_separation(task_list, no_polling_servers)
    et_tasks ← [ ]
    for task in task_list do:
        if task type equals ET:
            append task to et_tasks
    et_tasks_all_groups ← [ ]
    for i equals 1 to no_polling_Servers do:
        et_tasks_group ← [ ]
        for task in et_tasks do:
            if task seperation equals i:
                append task to et_tasks_group
        append et_tasks_group to et_tasks_all_groups

    total_duration_groups←zeros_list((length(et_tasks_all_groups)))

    for index,task_groups in enumerate(et_tasks_all_groups):
```

```
        for task in task_groups:
            total_duration_groups[index-1]←total_duration_groups[index-1]+task.duration
    et_mask_seperation0 ← [ ]
    for task in et_tasks:
        append task.seperation==0 to et_mask_seperation0

    et_tasks_seperation0← [ ]
    for task,bool in et_tasks,et_mask_seperation0:
        if bool:
            append task to et_tasks_seperation0

    for task in et_tasks_seperation0:
        index=index_minimum(total_duration_groups)
        et_tasks_all_groups[index].append(task)
        total_duration_groups[index] ← total_duration_groups[index] +task.duration
    return et_tasks_all_groups
```

6.4 Implementation of Extension 3

In section 6.1 we described the first EDP analysis implementation. This implementation uses the lower bound supply function ($lslbf$) and we also consider the worst possible latency (Δ), so it is a pessimistic way of evaluating the ET tasks schedulability. In this way, we are considering some solutions not able to schedule ET tasks when in fact they are able to. To avoid this, following extension 3 description, we use the schedule table to know the real supply function of each polling server. To do this, we developed the function *supply_poll_server* which is responsible to return the schedule table of each polling server (*supplymask*) . This way, we can implement the real supply function easily:

$$supply \mathrel{+}= supply\_mask[t]$$

Calculating the supply function this way,the demand function also changes, comparing to the implementation described in section 6.1:

$$H(t) = \sum_{\forall \tau_j \in \mathcal{T}^{ET}, p_j \geq p_i} C_j$$

With these small changes, we have a realistic EDP analysis, which will help to find more solutions able to schedule the ET tasks.

```
### pseudocode for supply_poll_server ###
""" Function responsible to return the  schedule table of the polling server
passed as argument"""

tt_schedule - TT tasks table schedule
poll server- value which represents the polling server studied
    supply_mask ← empty_list(size=(tt_schedule.size))
```

```
name ← "tPS"+poll_server
i←0
while i<tt_schedule.size do:
    supply_mask[i] ← (name == tt_schedule[i])
    i=i+1
end
return supply_mask
```

7 Simulated Annealing

7.1 Concept and general implementation

The simulated annealing concept is inspired by an analogy between the physical annealing process of metals and the problem of solving large combinatorial optimization problems. It is a variant of the hill climbing method and consists of a combination between a random walk through of the search space and a local search [4]. Neighbouring solutions are always accepted if they are an improvement. However the benefit of SA is that a temperature controlled function, as inspired by actual annealing, determines a random but decreasing chance to accept worse solutions thereby escaping local optima.

Below a generic implementation of simulated annealing is illustrated with pseudo code based on the implementation in [1]. The algorithm creates an initial solution i_{start} and temperature T_0 in an initialisation function. The algorithm then enters a loop and generates a new solution called j from the solution neighbourhood S_i. If the new solution is better it is always accepted. If the new solution is worse... Lastly a new temperature T_k is calculated. The loop then repeats until a stop-criterion is met. This could be time, number of iterations or a minimum temperature.

```
### pseudocode for simulated annealing ###
begin
    initialize(i_start,T_0)
    i = i_start
    repeat
        generate(j from S_i)
        if f(j) <= f(i)
            then i = j
        else
            if exp(f(i)-f(j) / T_k) > random[0,1)
                then i = j
        calculate_cooling(T_k)
    until stop-criterion
end
```

7.2 Our implementation

There are three parts of the generic implementation that are especially important. 1) The cooling function which controls the temperature change. 2) The cost function which determines the candidate solution cost and also is involved in the random chance of accepting inferior solutions. 3) The finding of new candidate solutions from the current solution neighbourhood through transforming relevant solution parameters. These topics are discussed below.

Temperature control

The cooling function used follows the formula shown below. The temperature is reduced iteratively from the original starting temperature and depends on the iteration number n and a

cooling factor a.

$$T_n = \frac{T_0}{(1 + a \cdot n)}$$

where

T_0 is the initial temperature

a is the cooling factor

n is the current iteration

The function is chosen so that the temperature decreases more rapidly initially before reducing exponentially before stabilising at a even level. This means that worse solutions have a higher chance being accepted at the beginning and a low even rate for the latter iterations.

By choosing different values for the cooling factor and the initial temperature it is possible to further optimise the algorithm. The initial temperature relates to the cost of a solution and should be a similar magnitude. This is why the solutions costs are normalised during simulated annealing. The closer the temperature is to the difference between the candidate and best solution costs the easier it is to control the function. For our final implementation the initial temperature is set to around $T_0 = 20$. The influence of the cooling factor a on the temperature is shown on figure 3 below. The smaller the cooling factor the slower the temperature drops and vice versa. In our final implementation the cooling factor is equal to around $a = 0.5$.

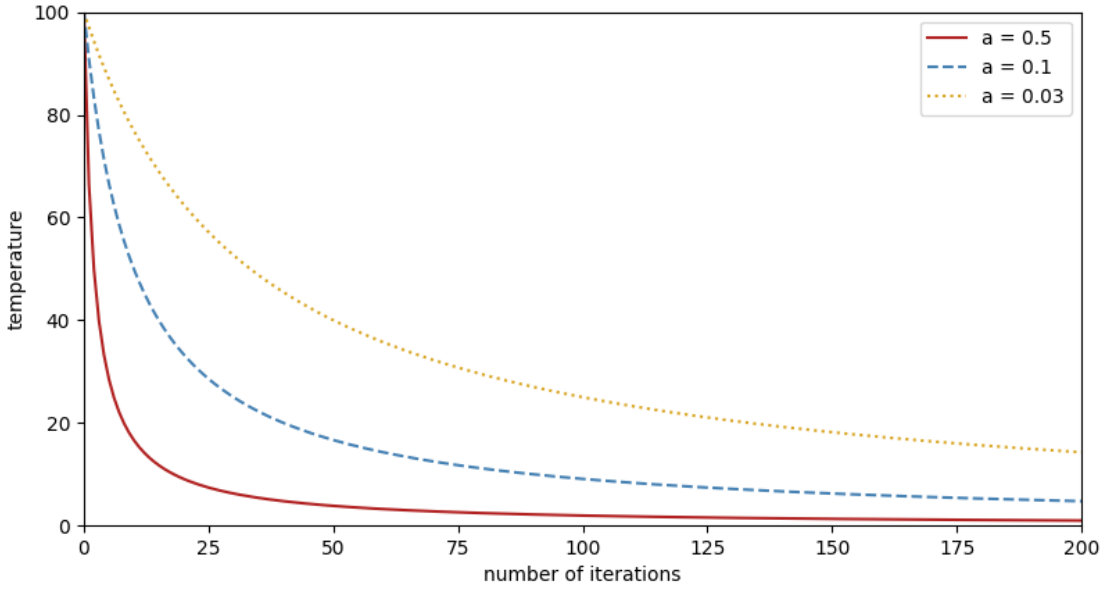


Figure 3: Effect of a factor on cooling function. Initial temperature $T_0 = 100$ and iterations $n = 200$.

Cost function

To compare the different candidate solutions and find the best one, following the project description, we decided to search for a solution where the average WCRT of all tasks (TT and ET) is minimized. The formula below shows how we calculate total cost for a proposed solution:

$$cost\_function = \frac{\sum_{\forall \tau j \in \mathcal{T}^{TT}} \cdot WCRT_{\tau j}}{len(\mathcal{T}^{TT})} + \frac{\sum_{\forall \tau j \in \mathcal{T}^{ET}} \cdot WCRT_{\tau j}}{len(\mathcal{T}^{ET})} \cdot (1 + 2 \cdot bool\_var \cdot coefficient)$$

bool_var represents the number of polling servers that can't successfully schedule the ET task set assigned

$coefficient$ determines the size of the penalty applied to solutions where ET tasks are not schedulable

Using the above formula a single value for optimising against is determine whilst punishing the solutions that are not capable of scheduling the ET tasks, but considering them as options for in our search for the best solution. The code section below shows a general implementation of our cost function:

```
### pseudo-code for cost_function ###
def cost_function(tt_wcrt, et_wcrt_groups, et_sched):
    if tt_wcrt and/or et_wcrt equals 0:
        return cost = 999999999999
    for item in tt_wcrt:
        tt_sum = tt_sum + item
    for group in et_wcrt_groups:
        for item in group:
            et_sum = et_sum + item
    cost = tt_sum/length(tt_wcrt) + et_sum*(1+2*et_sched*penalty)/length(et_wcrt_groups)
    return cost
```

In our implementation we only calculate the cost function of the current candidate solution if it is possible to schedule the TT tasks along with the proposed polling servers. Only then do we attempt to schedule the ET tasks and calculate the cost.

Candidate solution parameters

To find the new candidate solution in the neighbourhood of the current best solution we can vary two parameters; the polling server budget and period. We have reduced our search space by not considering the number of polling servers as a parameter to change. Therefore, the minimum number of polling servers, depending on the task set, are generated from the separation constraints. So, the ET tasks with the same separation value are assigned to the same polling server.

We define the budget and period variation maximum values and, using a random function we calculate a variation value for each of the new polling servers. Their new budget and period

are defined adding that assigned variation (positive or negative) to the current best solution values. When assigning new values to the period and budget it is sensible to create some constraints of which values these parameters can take. Firstly, to consider a valid period value, the hyperperiod must be fully divisible by the proposed value. In this way, the hyperperiod can be minimised taking into account the current test case data set.

In addition, we also only accept it a period value if it is greater than 2 as stated in the assumptions section (section 3.2). For the budget values, we consider the same constraint, it has to be greater than 0 and also lower than the period assigned to the polling server. When all the new budget and period values are defined, the polling servers are created and the tasks schedulability in this new candidate solution will be studied.

8 Algorithm example

Below follows a step by step example of the algorithm in action. This section follows the same structure as the algorithm overview on the figure shown in section 4. In the example a test case with 50 tasks (30 TT tasks and 20 ET tasks) called "taskset___1643188066-a_0.1-b_0.4-n_30-m_20-d_unif-p_2000-q_4000-g_1000-t_5___8___tsk.csv" from the "inf_10_40" folder is used. Below is an excerpt of the *task_list* object containing the information for all the tasks in the test case. This is the first step where data is read from the csv file into the algorithm after the initialisation of global variables and extra packages.

```
task_list = {'00' : {'deadline': 3000, 'duration' : 12, 'name' : 'tTT0',
             'period' : 3000, 'priority' : 7, 'seperation' = 0, 'type' : 'TT'},
             '01' : {'deadline': 2000, 'duration' : 7, 'name' : 'tTT1',
             'period' : 2000, 'priority' : 7, 'seperation' = 0, 'type' : 'TT'},
             ...
             '48' : {'deadline': 1328, 'duration' : 13, 'name' : 'tET14',
             'period' : 2000, 'priority' : 6, 'seperation' = 1, 'type' : 'ET'},
             '49' : {'deadline': 1136, 'duration' : 6, 'name' : 'tET3',
             'period' : 2000, 'priority' : 6, 'seperation' = 0, 'type' : 'ET'}}
```

Next the *task_list* is analysed for how to separate the ET tasks. First using the *task_seperation* function which determines the unique separation values as $[0, 1, 2]$ which means that a minimum of two polling servers are necessary. It also returns the number of ET task which is $n = 20$. Then with the *et_tasks_seperation* the task list is used to create a new task list where ET tasks are grouped by which separation value they are assigned. Immediately thereafter the priorities of each task are reassigned within each polling server. Notice that below tasks with separation value of 0 are split between the two polling servers for tasks with values of 1 and 2. This results in one server with 7 tasks and one with 13.

```
et_task_groups = {'0' : {'00' : {'deadline': 3720, 'duration' : 4, 'name' : 'tET1',
                          'period' : 4000, 'priority' : 6, 'sep.' = 1, 'type' : 'ET'}},
                          ...
                         {'06' : {'deadline': 1502, 'duration' : 73, 'name' : 'tET4',
                          'period' : 2000, 'priority' : 4, 'sep.' = 0, 'type' : 'ET'}},
                  '1' : {'00' : {'deadline': 3989, 'duration' : 162, 'name' : 'tET7',
                          'period' : 4000, 'priority' : 0, 'sep.' = 2, 'type' : 'ET'}},
                          ...
```

```
                    {'12' : {'deadline': 1200, 'duration' : 62, 'name' : 'tET10',
                     'period' : 2000, 'priority' : 4, 'sep.' = 0, 'type' : 'ET'}}}
```

Following this the two types of tasks are scheduled based on the default values for the polling serves and a candidate solution can be generated. The TT schedule is a list of length 12000 (the hyperperiod) and specifies which TT task is active for each tick. For the polling servers with the information shown as below the TT tasks are schedulable. The polling server information is stored as:

```
budget_poll_srv = [10,10]
period_poll_srv = [100,100]
```

From this we calculate the WCRT for the TT tasks which for these polling server properties results in a time of 7318. This information is stored as an array with length 32 with an item for each TT task. Then we move on to the ET tasks which are schedules in their assigned groups. For the initial solution the ET tasks are not schedulable with the polling servers being unable to meet the deadline of two tasks which are assigned to the server for separation value 1. The WCRT for the tasks that are schedulable totals to 25111. The computed cost is punished as describes in section 7 since the ET tasks are not schedulable and the solution receives a total cost of 1005779. WCRT for ET tasks is stored as shown below:

```
et_wcrt_groups = {'0' : {'00' : 3, '01' : 2328, '02' : 601, '03' : 1656, '04' : 3684,
                         '05' : Na, '06' : Na}
                  '1' : {'00' : 3465, '01' : 193, '02' : 1548, '03' : 2717, '04' : 1548,
                         '05' : 2717, '06' : 193, '07' : 488, '08' : 1548, '09' : 193,
                         '10' : 193, '11' : 488, '12' : 1548}}
```

Now a new candidate solution is generated based on the existing solution. The variable parameters are the period and budget. For the periods a max variation parameter is defined within which the period is able to vary. Within this bound a random integer number is generated and either added or subtracted to the current value. It also ensures that the hyper period is always evenly divisible by the new period. For the budget a similar approach is used where it is ensured that the budget is larger than 1 but smaller than the period. The max variation for the period is 50 and for the budget 40. Thus from our initial solution of $budget\_poll\_srv = [10, 10]$ and $period\_poll\_srv = [100, 100]$ the generated candidate is:

```
budget_poll_srv = [19,38]
period_poll_srv = [75,60]
```

Now the cost of the new candidate solution is calculated. As the ET tasks are now schedulable the cost is not punished and calculates as 7484. Since the cost is lower than the previous solution our new candidate solution is automatically accepted as the best solution. For the simulated annealing we now repeat these last few steps until the exit condition is met. In our case a specified time.

Part III

Evaluation and testing

9 Results

The optimization of timeline scheduling for time-triggered and event-triggered tasks with polling servers has been tested on 9 different task sets, varying in size and complexity. For the result section we have focused on one specific task set to showcase that the system works and how it performs, but for the evaluation all nine task sets are addressed. The 8 additional results from the test of the optimization algorithm are listed in appendix 2 and a example of the test case data can be seen in section 3.1.

All the task sets have been tested with an initial temperature and cooling rate, the number of polling servers is depending on the task set in hand and will be determined in the start, but there budget and period have some initial numbers. (See table 2). The test consist of running every task set through ten times (with different seeds for the random generators) to get a representative overview of the final output, in regards to the best solution cost, number of iterations and the numbers of servers, together with there corresponding budget and period.

Temperature	20
Cooling rate	0.5
Number of polling servers	(Depending on the task set)
Budget	20
Period	25

Table 2: Initial setup parameters

Detailed results from the ten iterations of task set ”1” is showcased in table 3. The last column in the table shows the calculated average probability of selecting a new solution even if it is worse than the current accepted solution. To summarize the data collected in table 3, we come to a average cost of 1626 with 104 iterations per run. In general the solutions converge towards polling servers with small budgets and periods meaning they appear in the TT schedule often but not for long.

Task set "1"

1643188013-a_0.1-b_0.1-n_30-m_20-d_unif-p_2000-q_4000-g_1000-t_5___0___tsk

	Cost	# Iterations	# Servers	Budget	Period	Avg. probability*
Run 1	1533	94	3	(4, 6, 23)	(20, 25, 80)	66.1%
Run 2	1894	85	3	(1, 2, 7)	(12, 8, 40)	52.5%
Run 3	1642	106	3	(28, 10, 21)	(96, 40, 120)	62.1%
Run 4	1507	112	3	(10, 1, 22)	(50, 3, 125)	62.3%
Run 5	1530	90	3	(1, 11, 11)	(4, 48, 50)	59.0%
Run 6	1677	131	3	(18, 7, 5)	(120, 16, 30)	59.3%
Run 7	1735	116	3	(14, 2, 24)	(120, 4, 150)	58.7%
Run 8	1521	119	3	(24, 10, 1)	(125, 30, 5)	57.3%
Run 9	1540	87	3	(13, 27, 8)	(75, 96, 25)	57.9%
Run 10	1679	102	3	(24, 6, 12)	(160, 40, 48)	50.1%

Table 3: Results from ten iterations of the first task set

*Average probability of acceptance for accepted worse solutions. (not the actual average probability of accepting a worse solution, this only takes into account cases when it is in fact selected)

The graph seen on figure 4, shows the results from task set 1. The Y-axis shows the accepted solution costs for the different iterations. The X-axis highlights the number of feasible solutions (i.e. schedulable) found in each of the tests. The graph also visually shows the simulated annealing function of accepting worse solutions, thereby exploring more than one solution neighbourhood. This can also be seen in all of the other task sets in appendix 2 - Result. The colours of the run indicate if the final cost is better than the initial (blue) or if the final cost is worse (orange). Dots indicate where the best overall solution for each run was found. Green lines indicate where the final solution is also the best overall (see appendices).

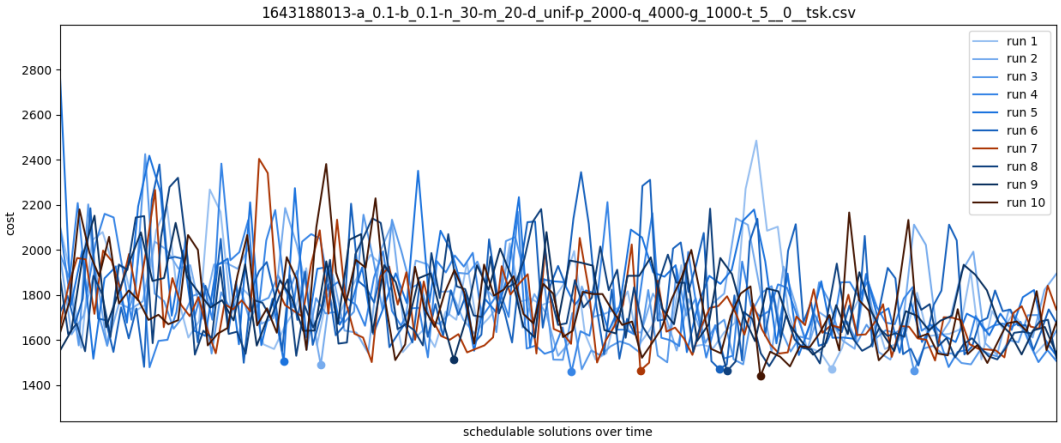


Figure 4: Accepted solutions costs over number of feasible solutions

For this task set the average improvement, between all 10 runs, from the initial cost to the final accepted cost was 317. The improvement from the initial cost to the minimum found cost during the run was 468.8. In general the variance in the minimum cost between all the runs is 498.8.

Next we look at the headline results for all ten of the test cases and how effective they are in finding an optimised solution. Table 4 below shows the initial cost, final cost and the minimal cost found in all 9 task sets. The reduction in cost $C_{init} - C_{final}$ from initial to final and from initial to minimum cost $C_{init} - C_{min}$ in are listed in the two last columns.

Task set	C_{init}	C_{final}	C_{min}	$C_{init} - C_{final}$	$C_{init} - C_{min}$
1	1942.8	1625.8	1474.0	317.0	468.8
2	2779.7	2096.3	1755.5	683.4	1024.2
3	1951.4	1643.0	1452.6	308.4	498.8
4	2569.7	18275	1679.5	742.2	890.2
5	4173.1	3434.9	3294.4	738.2	878.7
6	414899.3	3152.7	3062.6	411746.6	411836.7
7	11160.4	10513.7	9588.2	646.7	1572.2
8	5200.2	4180.3	4083.2	1019.9	1117.0
9	3212.0	3342.9	2989.9	-130.9*	222.1

Table 4: Cost change comparison across the 9 test cases

*Note that the negative value in the average difference indicates that on average the final solutions are worse than the initial solutions. This could be due to that this task set is particularly hard to schedule for. Additionally, the high initial costs for some solutions is due to the ET tasks not being schedulable. Therefore, the calculated cost is penalised and becomes very high. Generally we notice that for all text cases the final solution is better than the initial solution. However, the final solution is almost always worse than the minimum solution found during each run. In general between all the test cases the average reduction in cost from the initial solution to the minimum solution is 31.5%. For the result graphs for the other test cases the range of solution cost also narrows and becomes lower as the solution progresses.

10 Conclusion

After an analysis of the requirements of this project, we were able to design an optimization program that determines optimized solutions for the number of polling servers to use, as well as some characteristics about them, and which of the ET tasks can be scheduled within the polling servers, after performing some optimizations to their priorities and to the algorithm that manages them. For the TT tasks, firstly we checked if they are schedulable, using the Processor Demand Criterion and then we created an algorithm that finds the schedule, meaning that they finish before their deadlines. To fulfill the optimization objective, the task separation constraints are satisfied and the average worst-case response times of all tasks is minimized, in order to find the best solution possible, using the a simulated annealing based algorithm.

The decision to use the simulated annealing as the method to solve this optimization problem was based on the fact that we could have a better control of the solutions that the algorithm should accept by using the acceptance probability with a "temperature" decreasing at each step. In the beginning, the temperature allows almost all solutions to be accepted and then, when the temperature decreases gradually, it only accepted the improving solutions, as we expected.

After adding the extensions to the algorithms created, we concluded that we got an improvement in the test cases, meaning that the "weighted sum" obtained after the implementation of the extensions was smaller, and therefore the algorithms used are better than a random decision for the solution. Finally, in order to improve the solution provided, we came up with some features that could be done, such as:

- Running the program for longer periods of time, with low temperatures, so that our final solutions obtained have a better weighted sum than the minimum found solutions in the Results section, condition that it generally true for all the tens test cases reported.
- Using a different normalization function could result in better solutions, as the spectrum of results that we are defining can leave out some good solutions.
- The average probability for accepting worse solutions obtained in the simulated annealing algorithm should be similar for all test cases, so there is some room for improvement in this condition.
- The calculation of the number of polling servers used is only the minimum number required to schedule the all the ET tasks, but an improvement of this parameter would be to create an algorithm to assign variable number of polling servers, in order to verify if we get better schedule tables.
- Play with the separation value 0 ET tasks taking into account the parameters of the polling servers belonging to the candidate solution could improve the quality/cost of these candidates and also the number of feasible solutions per run.

11 Contribution

Table 5 is showing the personal contribution to the projects different tasks, both contribution to the code and the report.

	Alex BH.	Henrik N.	João M.	Tiago S.	Tomás E.
Algorithm parts					
tasks_parser			x		
divisble_random			x		
pdc			x		
edf_sim					x
edf					x
et_tasks_seperation				x	
supply_poll_srv				x	
et_schedule				x	
cost_function				x	
create_poll_srv	x	x			
simulated_annealing			x	x	
task_seperation	x	x			
priority_parser					x
main			x	x	x
Report parts					
Introduction and requirements	x				
Initial considerations and setup	x	x			
Algorithm overview	x				
EDF scheduling and Alg. 1					x
Extension 1			x		
EDP analysis and Algorithm 2		x		x	
Extension 2					x
Extension 3				x	
Simulated Annealing	x			x	
Algorithm example	x				
Evaluation and testing		x	x		
Diagrams and graphs	x	x			
Conclusion	x	x	x	x	x

Table 5: Contribution table

References

[1] Edmund K. Burke and Graham Kendall. *Search Methodologies*. Springer, 2 edition, 2014.

[2] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[3] Paul Pop. 02229: Systems Optimization Cyber-physical real-time systems.

[4] Franz Rothlauf. *Design of Modern Heuristics - Principles and application*. Springer, 2011.

Part IV

Appendices

Appendix 1 - Simplified code additional algorithm parts

```
### simplified code for task_parser ###
"""
Input format
path = "/Users/initials/folder"
file = "inf_10_10/taskset.csv"

Output format
task_list = { 'task1': {'key_A': 'value_A'},'task2': {'key_B': 'value_B'}}
"""
def task_parser(path, file):
    filepath = concatencate(path, file)
    read filepath to dataframe
    initialise task_list
    for each task in dataframe:
        append task to task_list
    return task_list
```

```
### simplified code for divisible_random ###
"""
Input format
a = 2
b = 100
n = 8

Output format
result = 24
"""
def divisible_random(a, b, n):
    if b - a < n:
        raise error "n is too large"
        exit
    result = random integer(a, b)
    while result-floor(result / n) is not equal to 0:
        result = random integer(a, b)
    return result
```

```
### simplified code for create_poll_srv ###
"""
Input format
```

```
budgets = [30, 55, 20]
periods = [3000, 2500, 4200]

Output format
return = { 'task1': {'key_A': 'value_A'},'task2': {'key_B': 'value_B'}}
"""
def create_poll_srv(budgets, periods):
    i = 0
    ps_list = []
    for budget and period in budgets and periods:
        ps_i = {name : tPS(i), duration : budget, period : period, type : TT, priority : 7,
                deadline : period, separation : 0}
        append ps_i to ps_list
        i = i +1
    return ps_list
```

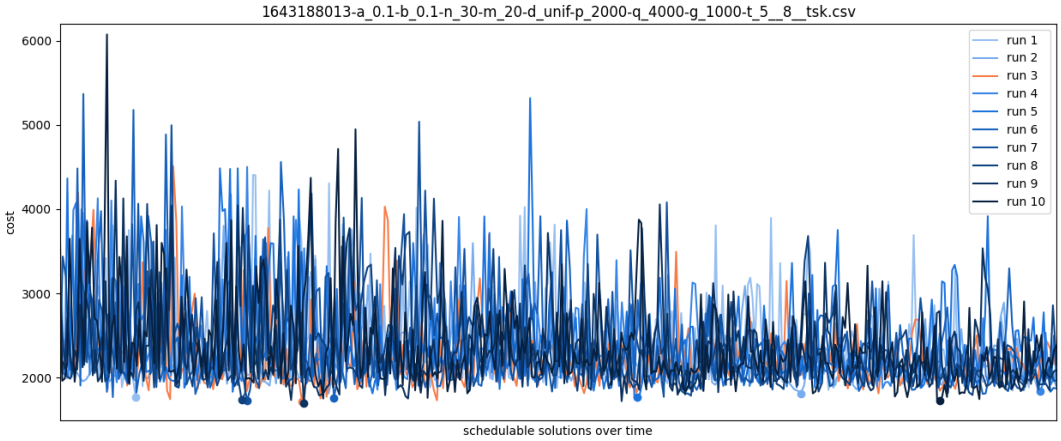
Appendix 2 - Results

Taskset "2"

1643188013-a_0.1-b_0.1-n_30-m_20-d_unif-p_2000-q_4000-g_1000-t_5__8__tsk

	Cost	# Iterations	# Servers	Budget	Period	Avg. probability*
Run 1	2156	434	2	(29, 4)	(48, 20)	72.0%
Run 2	2095	305	2	(10, 3)	(30, 10)	62.4%
Run 3	2370	326	2	(35, 2)	(96, 4)	63.2%
Run 4	1899	368	2	(2, 36)	(5, 120)	65.4%
Run 5	1985	393	2	(3, 2)	(6, 8)	61.1%
Run 6	1870	339	2	(36, 13)	(80, 48)	66.1%
Run 7	1964	331	2	(18, 1)	(40, 3)	62.6%
Run 8	2150	286	2	(46, 18)	(75, 80)	61.9%
Run 9	2091	393	2	(76, 6)	(150, 40)	64.5%
Run 10	2381	402	2	(29, 6)	(50, 48)	69.8%

Table 6: Results from ten iterations of the second taskset



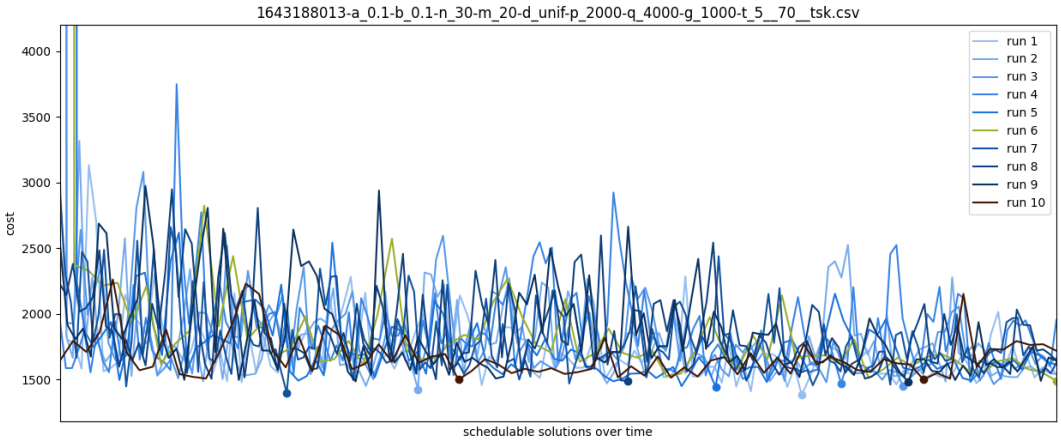
*This is the avg. probability of acceptance for worse candidate solutions accepted. (not the actual average probability of accepting a worse solution, this only takes into account cases when it is in fact selected, and the same applies for the remaining results tables)

Taskset "3"

1643188013-a_0.1-b_0.1-n_30-m_20-d_unif-p_2000-q_4000-g_1000-t_5___70___tsk

	Cost	# Iterations	# Servers	Budget	Period	Avg. probability*
Run 1	1537	138	3	(27, 1, 21)	(150, 4, 125)	53.9%
Run 2	1541	153	3	(1, 89, 5)	(6, 250, 25)	64.4%
Run 3	1665	143	3	(5, 5, 1)	(20, 16, 8)	57.7%
Run 4	1953	163	3	(10, 2, 1)	(50, 20, 3)	56.7%
Run 5	1613	116	3	(1, 3, 13)	(8, 12, 48)	58.6%
Run 6	1489	69	3	(30, 6, 1)	(125, 25, 4)	53.8%
Run 7	1627	181	3	(5, 12, 3)	(40, 50, 15)	62.0%
Run 8	1642	152	3	(3, 11, 4)	(25, 48, 16)	59.9%
Run 9	1645	129	3	(21, 26, 4)	(75, 75, 30)	51.7%
Run 10	1718	76	3	(1, 2, 59)	(3, 15, 240)	47.4%

Table 7: Results from ten iterations of the third taskset



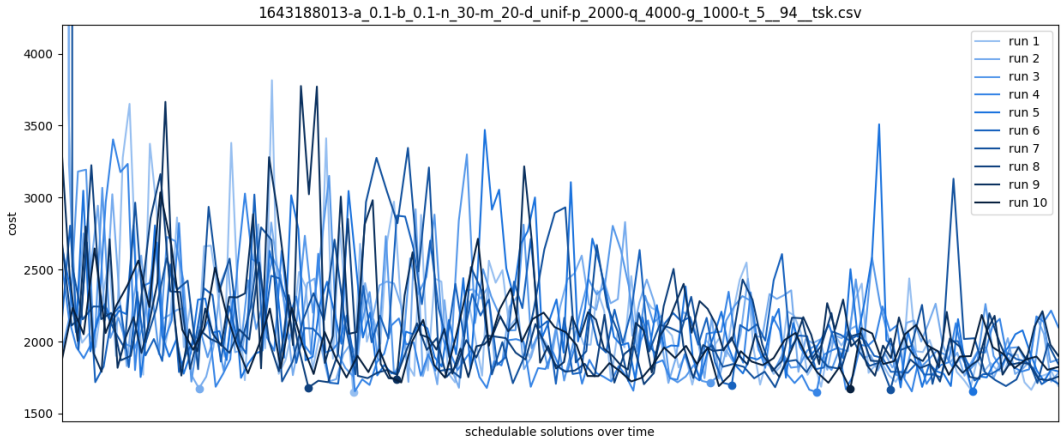
*This is the avg. probability of acceptance for worse candidate solutions accepted.

Taskset "4"

1643188013-a_0.1-b_0.1-n_30-m_20-d_unif-p_2000-q_4000-g_1000-t_5___94___tsk

	Cost	# Iterations	# Servers	Budget	Period	Avg. probability*
Run 1	1692	147	3	(5, 5, 40)	(25, 32, 160)	53.8%
Run 2	1783	138	3	(4, 4, 5)	(20, 40, 12)	62.2%
Run 3	1876	124	3	(19, 21, 5)	(50, 240, 24)	56.0%
Run 4	2099	137	3	(11, 5, 15)	(30, 25, 96)	50.5%
Run 5	1793	140	3	(25, 3, 10)	(96, 15, 30)	52.1%
Run 6	1707	120	3	(1, 53, 22)	(4, 250, 75)	47.2%
Run 7	1843	95	3	(16, 3, 22)	(60, 12, 80)	46.5%
Run 8	1757	102	3	(6, 20, 6)	(24, 96, 20)	51.5%
Run 9	1901	126	3	(2, 28, 5)	(10, 96, 20)	58.7%
Run 10	1824	92	3	(5, 5, 11)	(32, 24, 40)	45.9%

Table 8: Results from ten iterations of the fourth taskset



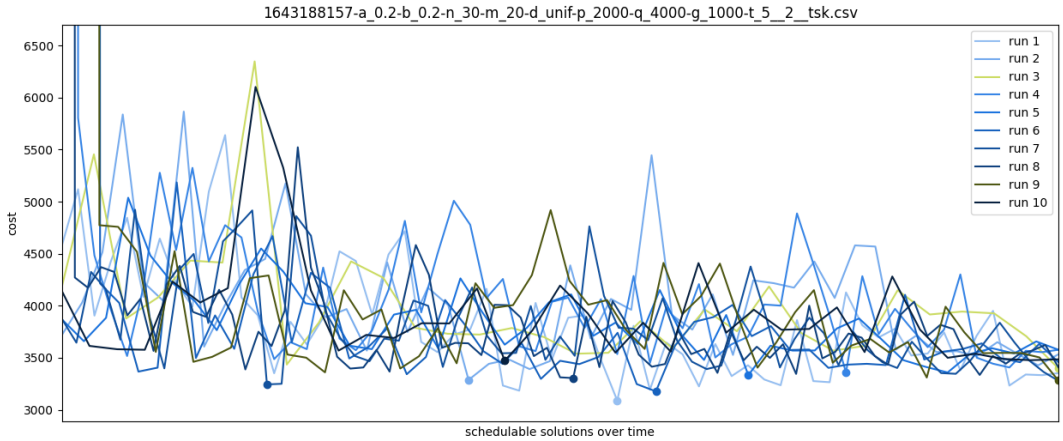
*This is the avg. probability of acceptance for worse candidate solutions accepted.

Taskset "5"

1643188157-a_0.2-b_0.2-n_30-m_20-d_unif-p_2000-q_4000-g_1000-t_5___2___tsk

	Cost	# Iterations	# Servers	Budget	Period	Avg. probability*
Run 1	3352	62	3	(35, 26, 1)	(100, 100, 15)	41.3%
Run 2	3533	50	3	(5, 28, 1)	(20, 80, 20)	56.6%
Run 3	3398	32	3	(1, 6, 1)	(3, 24, 15)	55.9%
Run 4	3577	61	3	(5, 5, 6)	(16, 20, 40)	43.8%
Run 5	3577	46	3	(1, 2, 2)	(4, 6, 25)	52.9%
Run 6	3458	51	3	(35, 10, 6)	(125, 30, 150)	50.7%
Run 7	3279	69	3	(41, 4, 12)	(120, 16, 160)	50.3%
Run 8	3409	76	3	(39, 21, 9)	(100, 120, 80)	57.7%
Run 9	3281	52	3	(40, 6, 2)	(125, 24, 20)	60.6%
Run 10	3485	37	3	(10, 2, 2)	(25, 10, 40)	50.1%

Table 9: Results from ten iterations of the fifth taskset



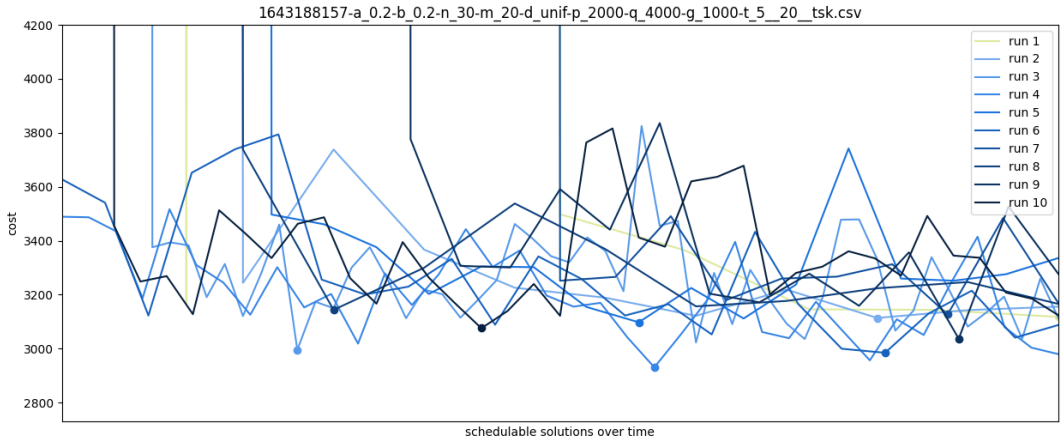
*This is the avg. probability of acceptance for worse candidate solutions accepted.

Taskset "6"

1643188157-a_0.2-b_0.2-n_30-m_20-d_unif-p_2000-q_4000-g_1000-t_5___20___tsk

	Cost	# Iterations	# Servers	Budget	Period	Avg. probability*
Run 1	3118	6	3	(7, 1, 8)	(30, 10, 20)	53%
Run 2	3156	10	3	(1, 6, 22)	(5, 80, 60)	64.8%
Run 3	3113	51	3	(21, 27, 9)	(100, 200, 24)	53.5%
Run 4	2980	38	3	(60, 14, 77)	(240, 150, 200)	43.5%
Run 5	3336	16	3	(16, 1, 35)	(80, 8, 80)	54.0%
Run 6	3088	24	3	(4, 3, 63)	(15, 20, 200)	52.6%
Run 7	3169	10	3	(1, 3, 9)	(5, 25, 24)	68.8%
Run 8	3166	10	3	(1, 1, 1)	(4, 8, 3)	70.0%
Run 9	3277	14	3	(8, 1, 23)	(32, 6, 75)	42.6%
Run 10	3124	37	3	(8, 5, 11)	(32, 80, 30)	66.3%

Table 10: Results from ten iterations of the sixth taskset



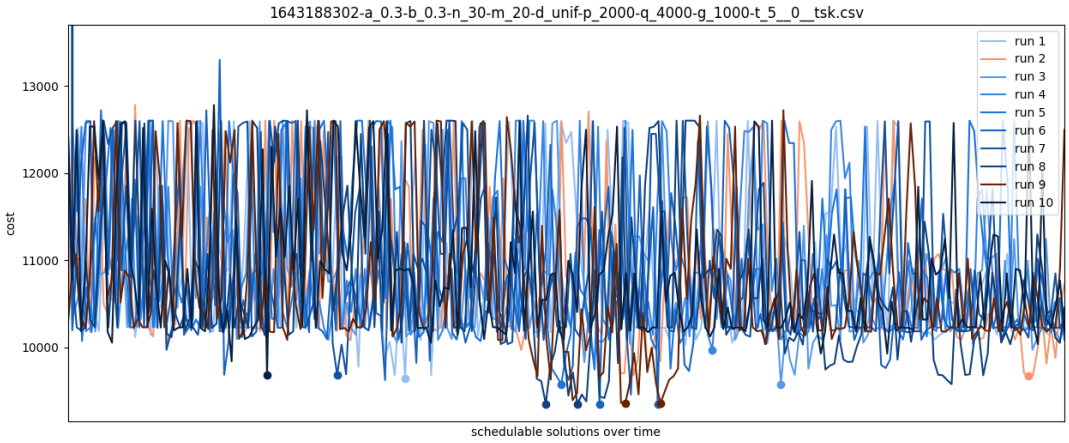
*This is the avg. probability of acceptance for worse candidate solutions accepted.

Taskset "7"

1643188302-a_0.3-b_0.3-n_30-m_20-d_unif-p_2000-q_4000-g_1000-t_5___0___tsk

	Cost	# Iterations	# Servers	Budget	Period	Avg. probability*
Run 1	10175	192	1	(20)	(30)	70,7%
Run 2	10728	223	1	(24)	(40)	74,0%
Run 3	10232	218	1	(2)	(3)	72,1%
Run 4	10224	218	1	(4)	(6)	75,2%
Run 5	10087	211	1	(27)	(40)	73,5%
Run 6	10320	223	1	(32)	(50)	75,6%
Run 7	10232	223	1	(2)	(3)	74,1%
Run 8	10456	221	1	(25)	(40)	74,3%
Run 9	12497	228	1	(10)	(20)	71,6%
Run 10	10186	226	1	(10)	(15)	72,0%

Table 11: Results from ten iterations of the seventh taskset



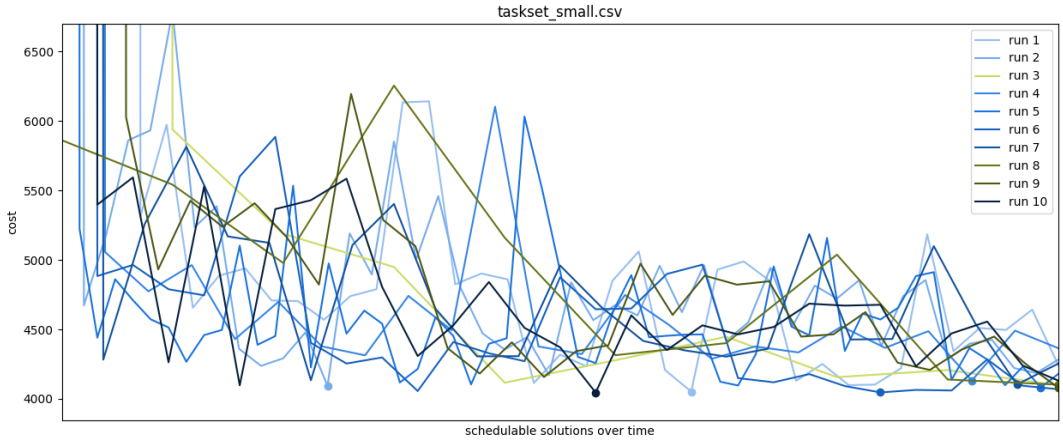
*This is the avg. probability of acceptance for worse candidate solutions accepted.

Taskset "8"

taskset_small

	Cost	# Iterations	# Servers	Budget	Period	Avg. probability*
Run 1	4242	36	3	(45, 1, 2)	(100, 8, 20)	58.5%
Run 2	4283	45	3	(22, 6, 1)	(50, 125, 16)	49.2%
Run 3	4098	9	3	(21, 2, 1)	(50, 16, 16)	51.7%
Run 4	4364	23	3	(3, 2, 1)	(10, 16, 10)	36.9%
Run 5	4180	56	3	(14, 1, 6)	(40, 8, 50)	57.7%
Run 6	4069	28	3	(16, 13, 8)	(40, 125, 80)	59.0%
Run 7	4254	24	3	(2, 3, 1)	(5, 20, 10)	40.2%
Run 8	4102	10	3	(4, 1, 1)	(10, 10, 8)	55.7%
Run 9	4080	30	3	(2, 2, 6)	(5, 20, 50)	59.6%
Run 10	4131	28	3	(2, 8, 6)	(5, 125, 50)	59.9%

Table 12: Results from ten iterations of the eighth taskset



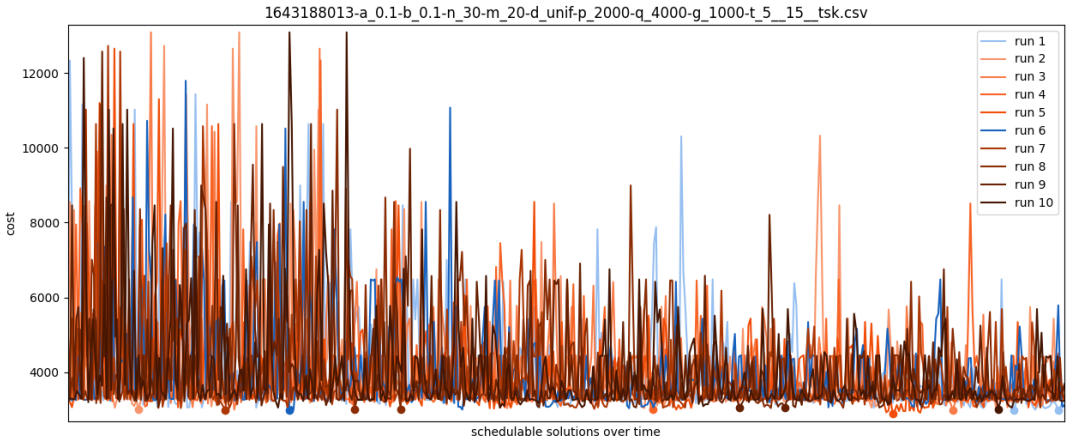
*This is the avg. probability of acceptance for worse candidate solutions accepted.

Taskset "9"

1643188013-a_0.1-b_0.1-n_30-m_20-d_unif-p_2000-q_4000-g_1000-t_5___15___tsk

	Cost	# Iterations	# Servers	Budget	Period	Avg. probability*
Run 1	3055	477	1	(58)	(80)	71.1%
Run 2	3285	466	1	(5)	(6)	72.7%
Run 3	3526	456	1	(2)	(3)	73.8%
Run 4	3254	478	1	(8)	(10)	70.2%
Run 5	3358	471	1	(14)	(20)	74.4%
Run 6	3110	491	1	(62)	(100)	73.1%
Run 7	3264	495	1	(15)	(20)	72.9%
Run 8	3685	456	1	(47)	(80)	71.6%
Run 9	3662	464	1	(5)	(8)	71.7%
Run 10	3230	437	1	(18)	(24)	72.2%

Table 13: Results from ten iterations of the ninth taskset



*This is the avg. probability of acceptance for worse candidate solutions accepted.