System Optimization Report

9/12 2022

Contents

1 Introduction

This optimization challenge is about configuring ADAS Applications with Time-Triggered and Event Triggered Tasks. Advanced Driver assistance systems (ADAS), provides driver assistance to e.g assited parking, lane switching, etc. Generally ADAS platforms are composed of heterogenous multu-core CPUs and Systems-on-chip (SoCs) of different safety levels and performance, that are interconnected by a real-time communication backbone. The projects target work is a *single computation element*, meaning a core in a multi-core SoC. Such a real-time component needs a scheduling policy. Scheduling is a method for assigning resource for activities that should be completed. The scheduling is handled and carried out by a *Scheduler* that implements such a policy that achieves the certain goals, these goals could be; minimizing latency, minimizing wait-time, and maximizing throughput.
The main approaches for making such a scheduler are: *time-triggered (TT)*, such as a non-preemptive static cylic scheduling, also called "timeline scheduling", and *event-triggered (ET)*, such as preemptive periodic scheduling. ET can be implemented using priorities that are fixed, Rate Monotonic (RM), or something dynamic like Earliest Deadline First (EDF).

In this project we consider TT tasks that are given are scheduled with *timeline scheduling*. However it is difficult to integrate sporadic event triggered (ET) tasks into a time-triggered system. Therefore we assume that ET tasks are integrated with TT tasks using *polling servers*[1].

The project aims to provide a design and implementation for optimization algorithms that determines an optimized solution for a given input. The input is a set of TT tasks, a set of ET tasks. TT tasks are periodic and consist of an infinite sequence of identical activities, referred to as *jobs* which are activated regularly with a constant *period*, making them *periodic*. The period, worst-case execution time (WCET) and deadline, is giving for each periodic TT task. ET tasks are however sporadic, which unlike periodic, can appear in from a minim inter-arrival time (minimum period). The minimum period, WCET and deadline as well as priority is given for ET tasks. All tasks, both ET tasks and TT tasks, are considered *preemptable*. This means that the tasks can be stopped for another task to run and then restarted to complete afterwards.

The solution for such input will contain; 1) The number of polling servers. 2) The period, budget and deadline for each polling server and 3) the sub-set of ET tasks within each polling server. For a solution to be valid, the TT tasks are also needed to be scheduable, meaning that the given found polling servers it should be possible to find a TT schedule such that all the TT tasks are scheduable.

The goal is to give an optimized soltuion such that; a) both TT and ET tasks are scheduable, meaning that they complete before their given deadline, b) ET task separation contraints are satisfied TODO secion. c) The Worst Case Response Time (WCRT) of all tasks is minimized. WCRT is the time it takes for a given task to finish after being 'released'.
There are many ways of optimizing scheduling, however in this project we have focused on the algorithms Hill Climbing[4] and Simulated Annealing[6]. Which includes the investigating in how to find good neighborhood solutions, as well as looking at the optimization which implementing these algorithms result in.
In the project we denote the set of TT and ET tasks as T^{TT} and T^{ET} respectfully. a given task is defined as τ_i with a some parameters: p_i being the task priority, C_i being the computation time, T_i being the period and D_i being the relative deadline.

1.1 Work distribution

3 people have worked on this projects, and everyone had equal participation in every aspect in the development. Our work procedure has to been to implement the different elements together as it was hard to split work up with 3 people. One might be the one programming while the oversee or help with implementation. The use of 'CodeTogether' has also been used, which allows all of us to code together in real-time.

[1] 02229 E22 Optimization Challenge: Configuring ADAS Applications with Time-Triggered and Event-Triggered Tasks

2 EDF simulation

2.1 Overview

The purpose of the EDF (Earliest deadline first) simulator algorithm is to create a schedule table for all the TT tasks. The EDF simulator takes an input of TT tasks, which include the polling servers. The output is a schedule table and the WCRTs for all the TT jobs.
To create a TT schedule table, we have to simulate up to the hyperperiod of TT tasks. To find such hyperperiod for all TT tasks we use the least common multiple of all the periods. Algorithm 7 implements such.

2.2 Algorithm 1 and 2 - EDF simulation

Algorithm 1: Scheduling TT tasks via EDF simulation

Some very important text
Input: Set of Time tasks and polling servers
Output: σ and WRCTs for TT Tasks (WRCt$_i$)

$T \leftarrow lcm(T_i | \forall T^{TT} \cup T^{poll})$;
$\forall \tau_i \in T^{TT} \cup T^{poll} : c_i \leftarrow D_i; r_i \leftarrow; WCRT_i \leftarrow 0$;
$t \leftarrow 0$;
while $t < T$ **do**
  foreach $\tau_i \in T^{TT} \cup T^{poll}$ **do**
    if $c_i \wedge d_i \leq t$ **then**
      return $\emptyset$; `// Deadline miss`
    end
    if $t\%T_i == 0$ **then**
      $r_i \leftarrow t$;
      $c_i \leftarrow C_i$;
      $d_i \leftarrow t + D_i$;
    end
  end
  if $[c_i = 0, \forall i \in T^{TT} \cup T^{poll}]$ **then**
    $\sigma[t] \leftarrow idle$
  else
    $\sigma[t] \leftarrow \tau_i = EDF(t, T^{TT} \cup T^{poll})$;
    $c_i \leftarrow c_i - 1$;
    if $c_i == 0 \wedge d_i \geq \wedge t - r_i \geq WCRT_i$ **then**
      $WCRT_i \leftarrow t - r_i$;
    end
  end
  $t \leftarrow t + 1$
end
if $[c_i > 0, \in T^{TT} \cup T^{poll}]$ **then**
  return $\emptyset$;
end
return $\sigma, WCRT_i$;

We started using this algorithm given in the assignment description. To test if it is possible to create a schedule table with the given time tasks. We found out that it was quite slow because it loops through the whole hyperperiod, and especially when we have to run this algorithm thousands of times in our simulated annealing. We also needed a way to see how close we are to a working table if the algorithm fails to create a table. Which right now it just returns null. So it is hard to know if get closer to a correct solution in an objective function. We made some adjustments and our new algorithm looks like algorithm 2.

Algorithm 2: New Scheduling TT tasks via EDF simulation

Input: Set of Time tasks and polling servers
Output: σ and WRCTs for TT Tasks (WRCt$_i$), $missingComputation$

$T \leftarrow lcm(T_i | \forall T^{TT} \cup T^{poll})$;
$\forall \tau_i \in T^{TT} \cup T^{poll} : c_i \leftarrow D_i; r_i \leftarrow; WCRT_i \leftarrow 0$;
$t \leftarrow 0$;
while $t < T$ **do**
  foreach $\tau_i \in T^{TT} \cup T^{poll}$ **do**
    if $c_i \wedge d_i \leq t$ **then**
      $missedComputation \leftarrow missedComputation + c_i$;
      $ci \leftarrow 0$;
    end
    $nextEvent \leftarrow min(nextEvent, nextRelease_i)$;
    $nextEvent \leftarrow min(nextEvent, nextDeadline_i)$;
    if $t\%T_i == 0$ **then**
      $r_i \leftarrow t$;
      $c_i \leftarrow C_i$;
      $d_i \leftarrow t + D_i$;
    end
  end
  if $[c_i = 0, \forall i \in T^{TT} \cup T^{poll}]$ **then**
    for $ttonextEvent$ **do**
      $\sigma[t] \leftarrow idle$
    end
  else
    $\tau_i = EDF(t, T^{TT} \cup T^{poll})$;
    $nextEvent \leftarrow min(nextEvent, c_i)$;
    for t *to* $nextEvent$ **do**
      $\sigma[t] \leftarrow \tau_i)$;
      $c_i \leftarrow c_i - 1$;
    end
    if $c_i == 0 \wedge d_i \geq \wedge t - r_i \geq WCRT_i$ **then**
      $WCRT_i \leftarrow max(t - r_i, WCRT_i)$;
    end
  end
  $t \leftarrow t + nextEvent$;
end
if $[c_i > 0, \in T^{TT} \cup T^{poll}]$ **then**
  $missedComputation \leftarrow missedComputation + c_i$;
end
return $\sigma, WCRT_i,\ missingComputation$;

Improved EDF

2.3 Complexity and Correctness

It can be seen that we use the *nextEvent* to see either when a new task is getting released or a task has computed until it has zero computation left. And in the end we skip to that time. We can do this because we know these are the only times a the table changes which tasks it is working on. So we will only iterate through the while loop as many times as a task gets added to the schedule table, instead of the whole hyperperiod.
And as seen on line 4, we add the computation has left after it's deadline to *missingComputation*, so we know how bad the schedule table failed. In the simulated annealing we want to minimize this to zero, and we can check if the table is schedulable, by checking if this value is zero.
Both version will have the same worst case complexity: $O(T \cdot size(T^{TT} \cup T^{poll}))$.

3 ET schedulability analysis using Explicit Deadline Periodic

As mentioned ET tasks are handled using polling servers in this project. There can be multiple polling servers containing multiple ET tasks for a solution, therefore we want to determine if such a polling server is schedulable, given its assigned ET tasks. Hence we want an algorithm to determine whether is polling server is schedulable for given a subset of ET tasks, as well as its worst-case response time. For now it is assumed that only one polling task τ_p handles the entire subset of ET tasks, where C_p and T_p has been predetermined.

3.1 Algorithm 3 and 4 - scheduleability of ET tasks

Algorithm 3: Schedulability of ET tasks under a given polling task

Input: Polling task budget C_p, polling task period T_p, polling task deadline D_p, subset of ET tasks to check T^{ET}
Output: $\{true, false\}$, responseTime

1 $\Delta \leftarrow T_p + D_p - 2 \cdot C_p$;
2 $\alpha \leftarrow \frac{C_p}{T_p}$;
3 $T \leftarrow lcm(T_i | \forall \tau_i \in T^{ET})$;
4 **foreach** $\tau_i \in T^{ET}$ **do**
5 $\quad t \leftarrow 0$;
6 $\quad worstCaseResponseTime \leftarrow responseTime \leftarrow D_i + 1$;
7 $\quad$ **while** $t \leq T$ **do**
8 $\quad\quad supply \leftarrow max(0, \alpha \cdot (t - \Delta))$;
9 $\quad\quad demand \leftarrow 0$;
10 $\quad\quad$ **foreach** $\tau_i \in T^{ET}$ *where* $P_j \geq P_j$ **do**
11 $\quad\quad\quad demand \leftarrow demand + \lceil \frac{t}{T_j} \rceil \cdot C_j$;
12 $\quad\quad$ **end**
13 $\quad\quad$ **if** $supply \geq demand$ & $t > 0$ **then**
14 $\quad\quad\quad responseTime \leftarrow t$;
15 $\quad\quad\quad worstCaseResponseTime \leftarrow max(responseTime, worstCaseResponseTime)$ *break*;
16 $\quad\quad$ **end**
17 $\quad\quad t \leftarrow t + 1$;
18 $\quad$ **end**
19 $\quad$ **if** $responseTime > D_i$ **then**
20 $\quad\quad$ **return** $false, worstCaseResponseTime$;
21 $\quad$ **end**
22 **end**
23 **return** $true, worstCaseResponseTime$;

Input could also be the polling server itself, and then extract the given values.

Algorithm 3 calculates the worst case distance, $\Delta = T_p + D_p - 2 \cdot C_p$. Figre 1 illustrates this worst case distance, with the next job finishing right before its deadline.

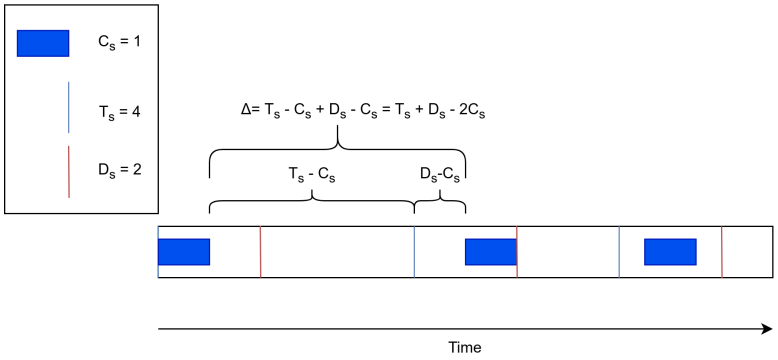


Figure 1: worst case distance illustration

This can however is can be improved. Since the schedule table is being constructed it is possible to determine the actual maximum distance of two jobs in a polling server, instead of the worst possible distance Δ. Hence an extension can be implemented to improve the algorithm, the extension is another algorithm to calculate Δ as explained in figure 1.

Algorithm 4: find max distance Δ

Some very important text
Input: polling server p, TT schedule table σ
Output: max distance

```
1  Δ ← 0;
2  currDistance ← 0;
3  computationTime ← p.computationTime;
4  isCountingDistance = σ[0] ≠ p ;          /* Flag to see if we are at desired polling server */
5  for i ← 0 to size(σ) by 1 do
6      if σ[i] == p then
7          if isCountingDistance then
8              maxDistance ← max(currentDistance, maxDistance);
9              currentDistance ← 0;
10             isCountingDistance ← false;
11         end
12         computationTime ← computationTime − 1;
13         if computationTime = 0 then
14             isCountingDistance ← true;
15             computationTime ← p.computationTime;
16         end
17     else
18         if isCountingDistance then
19             currentDistance ← currentDistance + 1;
20         end
21     end
22 end
23 if maxDistance = 0 then
       /* Failsafe if EDP is only scheduled for 1 job                              */
24     return T_p + D_p − 2 · C_p;
25 else
26     return maxDistance;
27 end
```

3.2 Complexity and correctness

Algorithm 4 calculates the distance from a polling server to the next time polling server is scheduled, hence it will calculate a more precise Δ value, representing the actual distance. We have implemented by using a flag to indicate when we first meet the polling server in the table, counting the steps until it is met again- However, if our distance is 0, a fail-safe is added in line 23 to ensure we then will return the more pessimistic way of calculating Δ.

Algorithm 3 have 2 loops, one for all ET Task and one for the size of the hyperperiod T. Therefore the complexity will be $O(size(T^{ET})^2 \cdot T$. However with the extension using algorithm 4, which runs in a loop in the size of the table, the complexity will be larger: $O(size(T^{ET})^2 \cdot T + size(\sigma))$.

4 Hill Climbing

4.1 Overview

Hill Climbing is a optimization algorithm which will keep trying to look at neighborhood solutions which are then evaluated and could perform better. The algorithm will then choose these solutions until no improvement can be found.
For Hill Climbing to work it has to start with a simple solution, and some defined neighbourhood. This would be a set of solutions which are similar but are constructed with a simple change. In our case, this could be a new set of polling servers. The algorithm cannot be guaranteed to find an optimal solution and can have bad performance, however hill climbing will often provide a solution very fast.

4.2 Algorithm 5 - Hill Climbing

Algorithm 5: Hill Climbing

Input: A set of event tasks s_0 (initial solution), amount of steps before finishing
Result: A set of polling servers, all with the same computation time, period and deadline

```
1  s_best ← s_current ← initial predefined solution;
2  o_current ← f(s);
3  period ← predefined;
4  CP_i ← 1;
5  while CP_i < period do
6  |   s_current.period ← CP_i;
7  |   o_new ← f(s.period);
8  |   if ∅_new < o_current then
9  |   |   s_best ← s_current;
10 |   |   o_current ← f(s_current);
11 |   end
12 |   CP_i ← CP_i + 1;
13 end
14 return
```

4.3 Complexity and correctness

Hill climbing will try and find a better neighbor for each iteration, selecting it if it is better. A neighbor is a set of polling servers with slightly different computation time. A better neighbor is determined by its *objective function* called *f()*. More about the objective function in section 5.8 It is hard to determine the complexity since hill climbing will finish when all the iteration steps has been executed. Hill climbing also makes use of EDF and EDF when using the *objective function*. The objective function executes EDP algorithm once, and the EDP algorithm is executed for each polling server, $n_{pollingServers}$. Hence the time complexity for hill climbing is $O(steps \cdot (O(EDF) + O(EDP) \cdot n_{pollingServers}))$.

5 Simulated annealing (SA)

5.1 Overview

Simulated Annealing is an improvement of Hill Climbing, seen in section 4. The key difference is that we selected a random solution from the neighborhood, not necessarily the best one. Then we will choose the best solution between the parent solution and the random solution, however SA can also randomly pick the worse solution. The purpose of this is to be able to escape a local neighborhood, and have to chance to find improvements in an other neighborhood where the initial solution might be worse, but a better solution might be in that area.
These worse solutions can be selected with a certain probability, this probability being less and less the further away we go from the original solution. Making it 'harder' to choose worse a solution the further away we get from the original solution.

5.2 Acceptance probability

Because we are doing simulated annealing, we have a chance to accept a solution that is worse than the previous, so we don't get stuck on a local best solution, and can find the global best one.
The probability is calculated:

$$p = e^{-\frac{\sigma}{co} \cdot \frac{\sigma}{t}}$$

Where δ is the difference in objective value between the new and current solution, our delta is always negative. The co is the current solution's objective value, and t is the temperature. By using the current objective value as a parameter, we can weigh the acceptance probability by the relative deterioration of value. Such that a solution which causes relatively significant degradation is less likely to be accepted.
The temperature is calculated:

$$t = 1 - \frac{currentStep + 1}{maxStep}$$

The temperature gets smaller the for every solution, we make, which decreases the overall probability. Also the smaller δ is the smaller is the probability. meaning it's harder to accept a solution, if it is almost as good as the current solution, and the probability gets smaller and smaller as the simulated annealing goes on. We multiply with $\frac{\sigma}{co}$ to scale the probability with the how drastic the change is. So a change from 10 to 15, has less chance to accept than a change from 1000 to 1005, even though δ is 5 for both cases. Figure 2 shows the graph for the acceptance probability, the lines represents σ values of -1, -3 and -50; worse than the current solution.

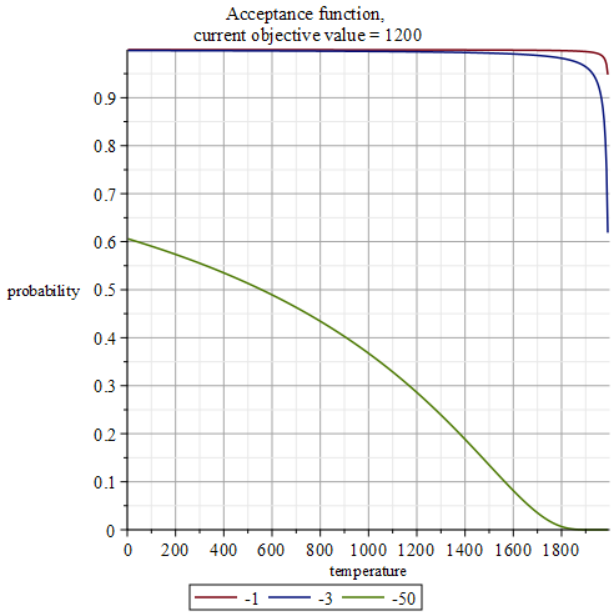


Figure 2: Acceptance probability function.

5.3 Algorithm 6 - Simulated Annealing

Algorithm 6: Simulated Annealing

Input: A set of random polling servers (initial solution)
Result: The best found solution after *maxSteps* steps

$s_{best} \leftarrow s_{current} \leftarrow solution_{initial}$;
$o_{best} \leftarrow o_{current} \leftarrow f(s_{current})$;
$steps \leftarrow 0$;
while $steps < maxSteps$ **do**
  $s_{new} = s_{current}.getNeighbor()$;
  $o_{new} = f(s_{new})$;
  $\delta = o_{current} - o_{new}$;
  if $\delta \geq 0$ **then**
    $s_{current} \leftarrow s_{new}$;
    $o_{current} \leftarrow o_{new}$;
    if $o_{new} < o_{best}$ **then**
      $s_{best} \leftarrow s_{current}$;
      $o_{best} \leftarrow o_{current}$;
  else if $\sigma < 0$ **then**
    $stepRatio \leftarrow sr(steps, maxSteps)$ $t \leftarrow temperatureF(stepRatio)$;
    $p \leftarrow ap(\delta, o_{current}, t)$;
    if $p > random(0, 1)$ **then**
      $s_{current} \leftarrow s_{new}$;
      $o_{current} \leftarrow o_{new}$;
  $steps = steps + 1$;
end
return s_{best};

5.4 Complexity and correctness

Where *f(s)* is an objective function[5.8] that takes a solution, and return a score for how good the solution is. And $p(\delta, t_i)$ is a probability function that takes, the change in score and how long the algorithm has run for, and return a change to accept the solution even if it is worth. We have made a stopping criteria that is a max number of steps, hence the complexity will be *O(maxSteps)*.

5.5 Defining a neighborhood

For our scenario, we need to provide a set of polling servers, which each has a collection of Event tasks and their own traits.
So we have made a class for our solutions called **PollingServerCollection** which most importantly has a **pollingServer** arraylist. And the functions **generateNeighbor()** and **static generateRandomSolution()**. For each polling server we can change the computation time, period, deadline and the subset of event tasks it has to to. Hence the neighbor of a giving polling server, would be such a polling server *but* with one of the parameters slightly changed.

5.6 Making initial solution

We make a random solution by calling the function *generateRandomSolution()* in the class **pollingServerCollection**, it takes a set of event tasks, a set of time tasks and numbers of polling servers we want to create. It creates a polling server for each separation requirement, and a number of polling servers giving as argument. From the set of time triggered tasks we find the hyperperiod using LCM, and create an array-list of numbers that are all factors of the hyperperiod. Each polling server then gets a random number from this list, as it's period. We do this because if the polling server got a completely random period we could end with an extremely long hyperperiod for our time table. We when set the deadline equal to period and computation to half. We give all the event tasks with a seperation requirement to the right polling server. And the other event tasks we distribute randomly between the servers.

5.7 Finding neighbor solution

We find a neighbor by doing a random mutation to a random polling server in a polling server collection. By calling **generateNeighbor()** on a polling server collection object. It has 25% change to either mutate a servers computation time, deadline, period or move an event task from one server to another. We change these values except the move task mutation, with a random number between 1 and a given max mutation step.

5.7.1 Mutate period

When mutating period we take a random server, and changes the period to another divisor of the hyperperiod. Every polling server have a field holding the index for the array list of divisors. And we change this index randomly to go up or down. If the period for a server goes down, we also change the deadline and computer computation time to keep the same ratio between it and the old period.

5.7.2 Mutate deadline

When we mutate the deadline we randomly choose to add or subtract the random amounts of steps between 1 and 5, to the deadline of a server. We make sure that the deadline is between 1 and the period.

5.7.3 Mutate computation time

The mutation of the computation time works the mutation of the deadline. We just make sure that the computation time is between 1 and the deadline.

5.7.4 Move Task

When moving a task from one polling server to another we find 2 random servers, from one of the servers we find a random task that doesn't have a seperation requirement, and moves it from server 1's list to server 2's.

5.8 Objective function

To see how good our solution is we use the objective function *objectiveFunction()* find in the **schedulor** class. It runs the EDF algorithm with all the time task and polling servers and the EDP algorithm on all polling servers. For the EDF we take the avg wcrt as the score, and if has any missed computation time on any of the tasks, it means the time table couldn't be made to work, so we multiply a penalty with the missing calculation time. Such that the less computation that can't be done gives a higher score.
We get a score from the EDP by taking the average wcrt from all the polling servers. And for every polling server that fails the EDP algorithm we multiply a penalty to it's WCRT.
In the end we add the EDF and the EDP scores, and that is the final score the objective function returns.

6 Least Common Multiple (LCM)

6.1 Overview

Least Common Multiple (LCM) is a helper algorithm, see 7, which finds the least common multiple between a set of numbers. That is the smallest number that is a multiple of the numbers in the set. The algorithm is used when we want to find the hyperperiod, which is the least common multiple of all the periods.

6.2 Complexity and correctness

The algorithm will run through each element in the set, trying to divide it down until all elements hit 1, while keeping track of the current lcm. When the counter is then equal to the size of the set (all elements has hit 1). Then the accumulated result will be returned. By doing this we calculate the lcm by using prime factorization, which factor each number, and gives it as a product of prime numbers. By multiplyng the primes with the highest factor we get the lcm.

The algortihm have an inner loop going through each element in the set of TT tasks, as well as an outer while loop. The other loop will at maximum run for $log_2(max(T_p \in T^{TT}))$, since a the largest number will have been divided down to 1 by then. Therefore the total complexity is $O(size(T^{TT}) \cdot log_2(max(T_p \in T^{TT}))$.

6.3 Algorithm 7 - LCM

Algorithm 7: Least Common Multiple (LCM)

Input: set of TT tasks
Output: Hyperperiod

```
1  divisor ← 2;
2  lcm ← 1;
3  while True do
4  |  counter ← 0;
5  |  divisible ← false;
6  |  foreach T_p ∈ T^TT do
7  |  |  if T_p == 0 then
8  |  |  |  return 0;
9  |  |  else if T_p < 0 then
10 |  |  |  T_p ← T_p · −1;
11 |  |  end
12 |  |  if T_p == 1 then
13 |  |  |  counter ← counter + 1
14 |  |  end
15 |  |  if T_p % divisor == 0 then
16 |  |  |  divisible ← true;
17 |  |  |  T_p ← T_p / divisor;
18 |  |  end
19 |  end
20 |  if divisible then
21 |  |  lcm ← lcm · divisor;
22 |  else
23 |  |  divisor ← divisor + 1
24 |  end
25 |  if counter == size(T^TT) then
26 |  |  return lcm;
27 |  end
28 end
```

7 Design Overview

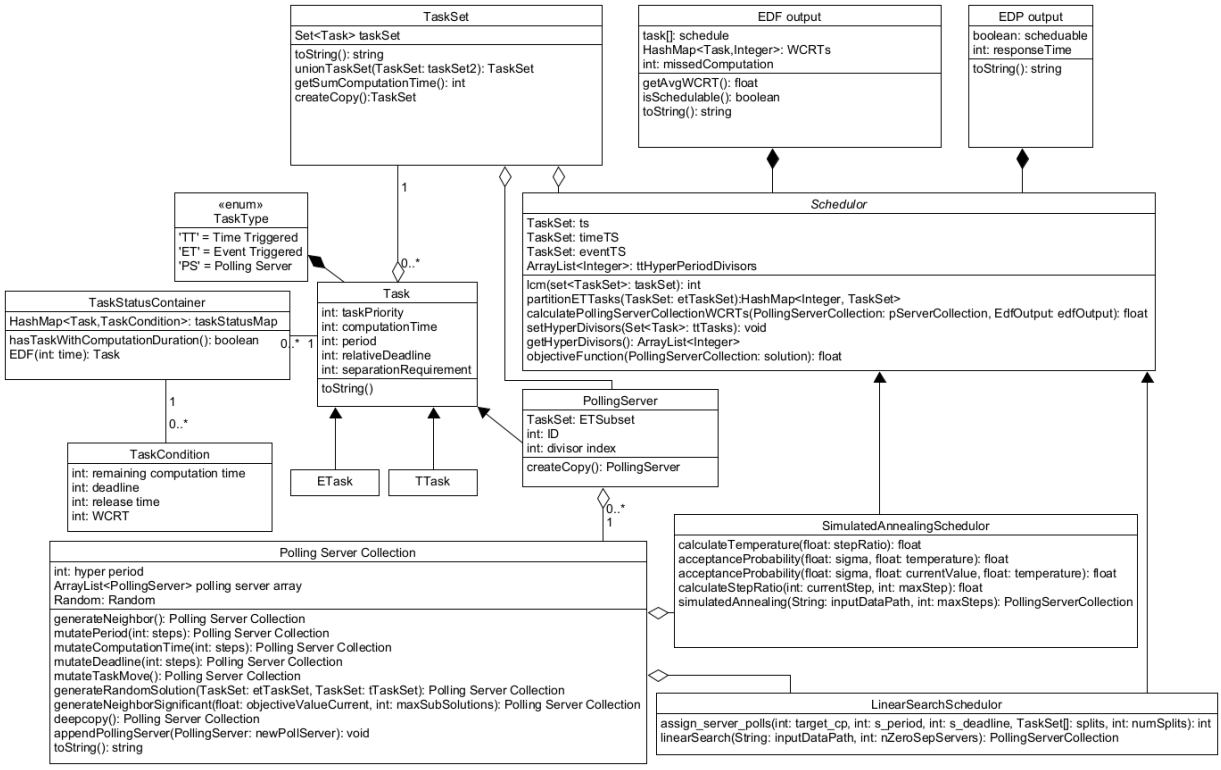


Figure 3: Class diagram

Figure 3 shows the class diagram of out implementation. It shows how we have implemented ET tasks, TT tasks and polling server. It also shows how we have made classes to handle set of tasks, as well as custom classes for outputs for our algorithms. As it can be seen on the class diagram, a polling server is an extension of a task, but is also taking in a TaskSet as a parameter. As a polling server can be seen as a TT task, but will contain ET tasks. Aswell as both *SimulatedAnnealingScheduler* and *LinearSearchScheduler* (Hill Climbing) inheriting the abstract Scheduler class.

8 Experiments

In this section we will explain the various experiments we have done regarding our different algorithms, and the overall problem to find an optimized solution.

8.1 Continuous Evaluation

During development, we continuously evaluated the modifications by running the program on the same dataset. Because of this, we could quickly notice when any critical mistakes were made. However, due to the stochastic nature of the heuristics, we were unable to notice any 'correctness' problems of the type that results in a valid, but bad solution.

Notably, we had a bug which caused us to not be able to change the computation time of the polling servers, and would instead wrongly modify the deadline. This issue persisted for quite a long time, until we noticed that manually choosing a good initial solution would very quickly be modified to obtain penalty. This lead us to investigate the segment of code responsible for generating neighboring solutions, and finally find the

source of disruption.

We also used this continuous evaluation of our development when fine-tuning the acceptance probability function; and for even better insight, we employed a symbolic mathematics program in order to graph out possible functions and find suitable coefficients, as seen in 2.

8.2 Hill Climbing Vs. Simulated Annealing

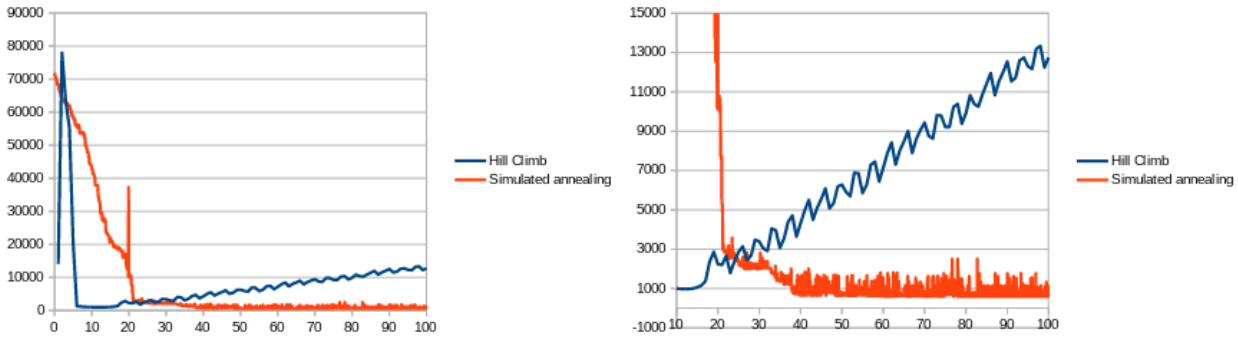
We initially established a hill climbing approach to the problem, in order to get a fundamental understanding of the task at hand. This hill climbing approach creates an initial solution using a predefined period, and a deadline and then modifies the computation by gradually increasing it each iteration. These properties are assigned for all servers - no matter which value of separation requirement they're handling.

The reason we only modify a single property is to reduce the size of the solution space; even modifying two properties will produce a solution space of 2 dimensions we have to search through to find the optimal solution. Thus, our implementation of hill climbing is a relatively simple method, so we wanted to compare the results with Simulated annealing, a more complex heuristic.

It turns out that, since all our polling servers in Hill Climbing use the same properties, it severely affects our ability to find a good solution. As in the case that a polling server handles tasks with separation requirement != 0, it might only contain a few tasks but still use as much computation as a polling server handling one third of the event tasks. This because a difficult task of balancing the universal computation time of all servers between those with much responsibility, and those with little responsibility.

In this regard, Simulated Annealing is more refined. Even though it does not make any promises of search through an a complete (but delimited) solution space like Hill climbing, it instead has the ability to modify all the properties, moving towards generally favorable solutions. For example, our simulated annealing solution enables the servers handling tasks with separation requirement of 0 to swap their tasks. And have individual computation times; as such, it avoids one of the prominent issues of hill climb. In other words, it has a larger range of solution space than hill climbing.

Figure 4: Hill climbing vs SA objective values through iterations



2

8.3 Results

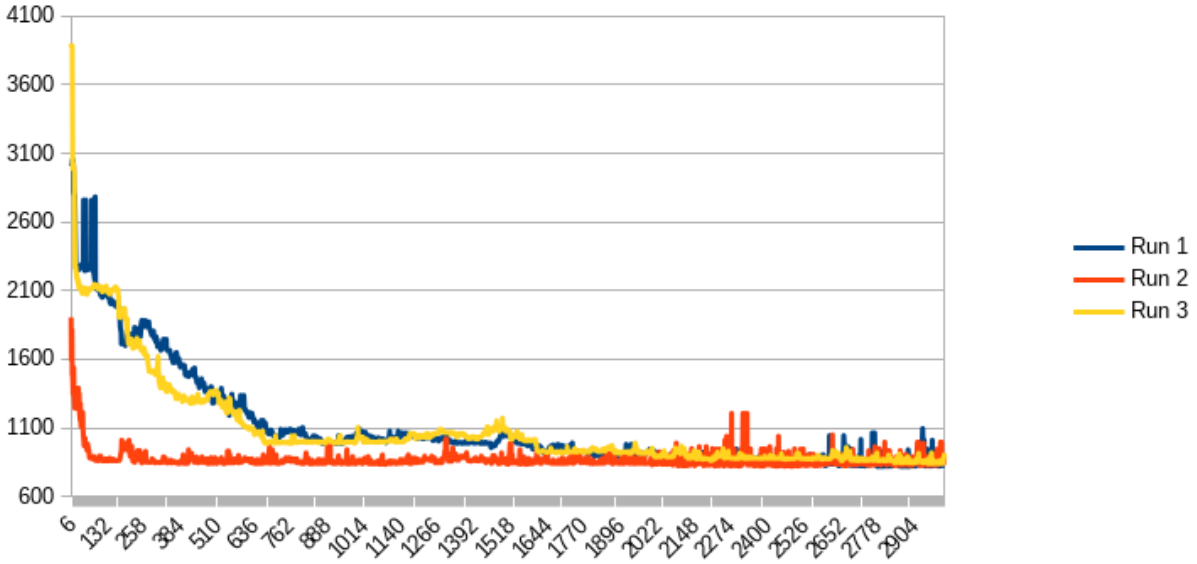
We experimented first with data set that have no separation and with only one server. We tested on the task set *taskset__1643188013-a_0.1-b_0.1-n_30-m_20-d_unif-p_2000-q_4000-g_1000-t_5__0__tsk.csv*. With 3 different runs with starting pollingServers:
{C; T; p; D}

[2]The x axis is percent of total iteration, hill climb has 100 iterations, while simulated annealing has 2000 iterations

1) {200;1000;7;1000;}
2) {20;100;7;100;}
3) {240;1200;7;1200;}

We have made a line graph showing the objective value of each created solution, for 3000 iterations. The

Figure 5: Objective value with 1 server no separation



best solutions found was:
1) {12;20;7;15;} with objective score: 819.161 after 2768 iterations.
2) {14;24;7;24} with objective score: 821.065 after 2448 iterations.
2) {18;30;7;20} with objective score: 824.71 after 2597 iterations.

So run one found the best solution but there is only a small difference for all of them. They all seems to find a solution that is very similar.
It can also be seen that run 3 has a very high value at the start, that is because it gets a EDP penalty, because the penalty gets multiplied, with the wcrt for the EDP output, it quickly finds a solution that dosen't have a penalty.
We don't get the same solution every time proving that simmulated annealing doesn't always find the best solution. But it looks like we find a good search space for finding for a good optimised solution.

Only one polling server is very simple, case so we expanded to use a test with separation requirement, and having 4 starting polling servers with separation requirement 0. We ran task set *taskset__1643188013-a_0.1-b_0.1-n_30-m_20-d_unif-p_2000-q_4000-g_1000-t_5__0__tsk*, and the results are:

Run1					
Randomstartsolution				Result	
Start	{c,T,p,d,sep}	EventTasks		{c,T,p,d,sep}	EventTasks
Pollingserver1	{2;25;7;22;1}	;tET3;27;3000;ET;2;2406;1 ;tET12;14;4000;ET;0;2968;1		{3;32;7;7;1}	;tET3;27;3000;ET;2;2406;1 ;tET12;14;4000;ET;0;2968;1
Pollingserver2	{33;80;7;66;2}	tET4;25;3000;ET;0;2998;2		{4;50;7;6;2}	;tET4;25;3000;ET;0;2998;2
Pollingserver3	{30;150;7;150;0}	;tET11;12;3000;ET;2;2492;0 ;tET8;30;2000;ET;3;1774;0 ;tET10;21;2000;ET;6;1220;0 ;tET7;3;4000;ET;2;2233;0		{3;100;7;70;0}	;tET9;2;2000;ET;6;1197;0
Pollingserver4	{9;48;7;48;0}	;tET9;2;2000;ET;6;1197;0 ;tET15;22;3000;ET;1;2697;0 ;tET14;1;2000;ET;5;1340;0 ;tET13;21;3000;ET;3;1922;0		{1;24;7;3;0}	;tET7;3;4000;ET;2;2233;0
Pollingserver5	{20;100;7;100;0}	;tET6;9;3000;ET;2;1951;0 ;tET5;4;3000;ET;5;1623;0 ;tET19;9;3000;ET;2;2437;0 ;tET18;11;2000;ET;6;1202;0		{4;125;7;81;0}	;tET0;3;3000;ET;2;2395;0
Pollingserver5	{19;96;7;96;0}	;tET0;3;3000;ET;2;2395;0 ;tET17;10;2000;ET;5;1627;0 ;tET2;13;2000;ET;5;1761;0 ;tET16;37;4000;ET;2;2516;0 ;tET1;6;2000;ET;5;1617;0		{25;100;7;84;0}	;tET8;30;2000;ET;3;1774;0 ;tET11;12;3000;ET;2;2492;0 ;tET10;21;2000;ET;6;1220;0 ;tET15;22;3000;ET;1;2697;0 ;tET14;1;2000;ET;5;1340;0 ;tET13;21;3000;ET;3;1922;0 ;tET6;9;3000;ET;2;1951;0 ;tET5;4;3000;ET;5;1623;0 ;tET19;9;3000;ET;2;2437;0 ;tET18;11;2000;ET;6;1202;0 ;tET17;10;2000;ET;5;1627;0 ;tET16;37;4000;ET;2;2516;0 ;tET2;13;2000;ET;5;1761;0 ;tET1;6;2000;ET;5;1617;0

Final, best Objective Value is: 673.0278
Improvement from original solution is: -92071.914
Best server iteration: 2286

Run2				
Randomstartsolution			Result	
Start	{c,T,p,d,sep}	EventTasks	{c,T,p,d,sep}	EventTasks
Pollingserver1	{2;12;7;5;1}	;tET3;27;3000;ET;2;2406;1 ;tET12;14;4000;ET;0;2968;1	{1;10;7;4;1}	;tET3;27;3000;ET;2;2406;1 ;tET12;14;4000;ET;0;2968;1
Pollingserver2	{7;8;7;8;2}	tET4;25;3000;ET;0;2998;2	{1;12;7;5;2}	;tET4;25;3000;ET;0;2998;2
Pollingserver3	{8;40;7;40}	;tET11;12;3000;ET;2;2492;0 ;tET8;30;2000;ET;3;1774;0 ;tET10;21;2000;ET;6;1220;0 ;tET7;3;4000;ET;2;2233;0	{1;24;7;8;0}	;tET0;3;3000;ET;2;2395;0
Pollingserver4	{5;25;7;25;0}	;tET9;2;2000;ET;6;1197;0 ;tET15;22;3000;ET;1;2697;0 ;tET14;1;2000;ET;5;1340;0 ;tET13;21;3000;ET;3;1922;0	{1;24;7;12;0}	;tET9;2;2000;ET;6;1197;0
Pollingserver5	{48;240;7;240;0}	;tET6;9;3000;ET;2;1951;0 ;tET5;4;3000;ET;5;1623;0 ;tET19;9;3000;ET;2;2437;0 ;tET18;11;2000;ET;6;1202;0	{1;100;7;62;0}	;tET14;1;2000;ET;5;1340;0
Pollingserver5	{48;240;7;240;0}	;tET0;3;3000;ET;2;2395;0 ;tET17;10;2000;ET;5;1627;0 ;tET2;13;2000;ET;5;1761;0 ;tET16;37;4000;ET;2;2516;0 ;tET1;6;2000;ET;5;1617;0	{38;150;7;121;0}	;tET11;12;3000;ET;2;2492;0 ;tET8;30;2000;ET;3;1774;0 ;tET10;21;2000;ET;6;1220;0 ;tET7;3;4000;ET;2;2233;0 ;tET15;22;3000;ET;1;2697;0 ;tET6;9;3000;ET;2;1951;0 ;tET13;21;3000;ET;3;1922;0 ;tET5;4;3000;ET;5;1623;0 ;tET19;9;3000;ET;2;2437;0 ;tET18;11;2000;ET;6;1202;0 ;tET17;10;2000;ET;5;1627;0 ;tET16;37;4000;ET;2;2516;0 ;tET2;13;2000;ET;5;1761;0 ;tET1;6;2000;ET;5;1617;0

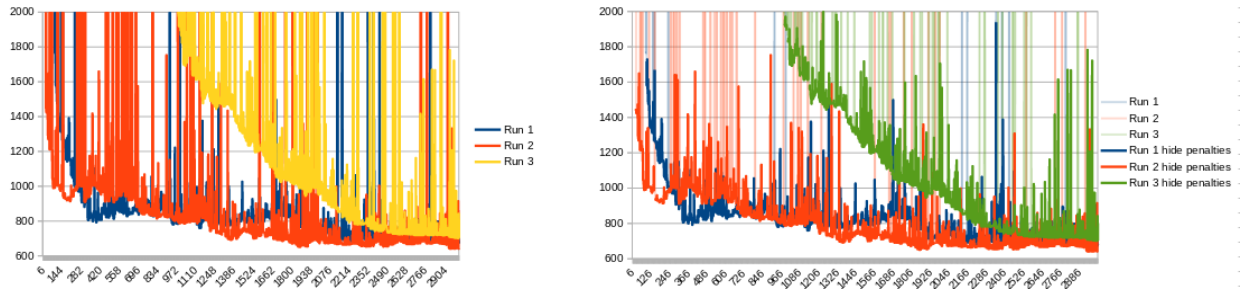
Final, best Objective Value is: 641.27783
Improvement from original solution is: -227040.47
Best server iteration: 2936

Run2				
Randomstartsolution			Result	
Start	{c,T,p,d,sep}	EventTasks	{c,T,p,d,sep}	EventTasks
Pollingserver1	{316;400;7;400;1}	;tET3;27;3000;ET;2;2406;1 ;tET12;14;4000;ET;0;2968;1	{13;150;7;122;1}	;tET3;27;3000;ET;2;2406;1 ;tET12;14;4000;ET;0;2968;1
Pollingserver2	{83;250;7;166;2}	tET4;25;3000;ET;0;2998;2	{18;200;7;86;2}	;tET4;25;3000;ET;0;2998;2
Pollingserver3	{1;5;7;5;0}	;tET11;12;3000;ET;2;2492;0 ;tET8;30;2000;ET;3;1774;0 ;tET10;21;2000;ET;6;1220;0 ;tET7;3;4000;ET;2;2233;0	{1;5;7;5;0}	;tET8;30;2000;ET;3;1774;0 ;tET11;12;3000;ET;2;2492;0 ;tET7;3;4000;ET;2;2233;0 ;tET10;21;2000;ET;6;1220;0 ;tET15;22;3000;ET;1;2697;0 ;tET13;21;3000;ET;3;1922;0 ;tET6;9;3000;ET;2;1951;0 ;tET5;4;3000;ET;5;1623;0 ;tET19;9;3000;ET;2;2437;0 ;tET18;11;2000;ET;6;1202;0 ;tET17;10;2000;ET;5;1627;0 ;tET16;37;4000;ET;2;2516;0 ;tET0;3;3000;ET;2;2395;0 ;tET1;6;2000;ET;5;1617;0
Pollingserver4	{6;30;7;30;0}	;tET9;2;2000;ET;6;1197;0 ;tET15;22;3000;ET;1;2697;0 ;tET14;1;2000;ET;5;1340;0 ;tET13;21;3000;ET;3;1922;0	{1;48;7;3;0}	;tET14;1;2000;ET;5;1340;0
Pollingserver5	{3;16;7;16;0}	;tET6;9;3000;ET;2;1951;0 ;tET5;4;3000;ET;5;1623;0 ;tET19;9;3000;ET;2;2437;0 ;tET18;11;2000;ET;6;1202;0	{1;15;7;7;0}	;tET2;13;2000;ET;5;1761;0
Pollingserver5	{4;24;7;24;0}	;tET0;3;3000;ET;2;2395;0 ;tET17;10;2000;ET;5;1627;0 ;tET2;13;2000;ET;5;1761;0 ;tET16;37;4000;ET;2;2516;0 ;tET1;6;2000;ET;5;1617;0	{1;32;7;2;0}	;tET9;2;2000;ET;6;1197;0

Final, best Objective Value is: 641.27783
Improvement from original solution is: -227040.47
Best server iteration: 2936

We also made a graph for the objective value for all three runs, the first graph shows that we have some really high values, when we apply either a EDF or EDP penalty, which makes the graph hard to read. So in the second graph we removed the values that was caused by penalties, which gives us a clear image of how the objective value gets smaller and smaller.

Figure 6: Objective value with 4 server and separation



We can again see that all of them finds a viable solution, and all of them approaches the same objective value. And a solution that looks alike. We also experimented with giving the start solution some more randomness when creating polling servers. It seems like our algorithm seems to favor making one server that holds all the event task, even though we start spreading them out evenly. We also experimented for finding

a good penalty score. Here it worked well with $20 \cdot wrct$ for EDP penalty, and $20 \cdot missedComputaiontime$ for EDF penalty.

8.4 Discussion

We have implemented a working version of simulated annealing that finds a given amount of polling servers such that we are able to create a schedule table where no time or event task misses the deadline, and because it wants to minimize the objective value it, tries to find the solution that has the lowest worst case response time. However do still need to manually input how many polling servers we want to create, a future extension could be a new mutation where it added and removed servers at a random change.

Our method is also based on a very random approach, a possible approach could be to implement dynamic mutations chance. The chances for our mutations are split evenly through the whole process. By changing them dynamically while the algorithm is running, it would make more educated guesses that could reduce the amount of iteration it takes before finding the best solution, e.g if a server fails EDP it could increase the odds of changing this server, or have a better chance to decrease the period or increase computation.

We have sometimes run into a run where we get stuck with a server that fails EDP. By expanding our solution such that it resembles tabu search, could solve this. We would save the solution we have made in the neighborhood `N(s)` and when we have discovered all the neighbors in the neighborhood for solution s, we would jump to a new neighborhood $N^*(s)$. We already have the function *generateRandomSolution()* we could use to jump to a new neighborhood. It would also decrease the amount of iterations needed since we stop visiting solutions we already have tried. The drawback is that the program will use more memory since we have to save previous found solutions.

To speed up the execution of our program, we can also implement 'delta evaluation' to some degree. Delta evaluation is the strategy of only evaluating what has changed since previous iteration.
In our instance, we do not need to evaluate EDP for each polling server; only the newly modified polling server. By reducing the running time of a scheduling algorithm, the industry can execute a greater number of iterations, potentially finding an even better solution. As this subject has dealt with pre-computed schedules to be used repeatedly in physical systems, an improved solution can have a drastic effect on the performance. Alternatively, we could investigate a 'genetic algorithm', which are population based metaheuristic. By having a population (a solution set), we choose 2 current solutions to serve as parents, and generate a new solution as a mix between the parents. The selection of parents can be weighed in many ways, but often a good solution corresponds to higher likelihood for selection. This uses the concept of evolution to create a final improved solution that hopefully has the best traits of the initial population.
The genetic algorithms are determined by a number of properties, such as selection method, size and generation of initial population, mutation method and more. Thus, it can be fine-tuned to suit a subject favorably, in terms of use - though it requires extensive analysis.

9 Conclusion

In this project we have been solving the Optimization Challenge: Configuring ADAS Applications with Time-Triggered and Event-Triggered Tasks.
We have made a program written in java that takes a task set of time triggered tasks and event triggered tasks, and a given number of polling servers. It then finds a optimized solution for what values and event tasks the polling servers should have, such that a schedule table can be created, with the smallest overall worst case response time, using simulated annealing. We have implemented extensions to the EDF and EDP algorithm such that the EDF runs faster and gives us a useful value if it fails, and the EDP uses the schedule from the EDF so it doesn't use a pessimistic approach.
We then used the EDF and EDP algorithm to create a weighted objective function, that also penalizes wrong solutions, to compare solutions to the problem.
We have discussed what is good and what could be done to improve our program. We have looked at different methods, and learned a lot about metaheuristic procedures.