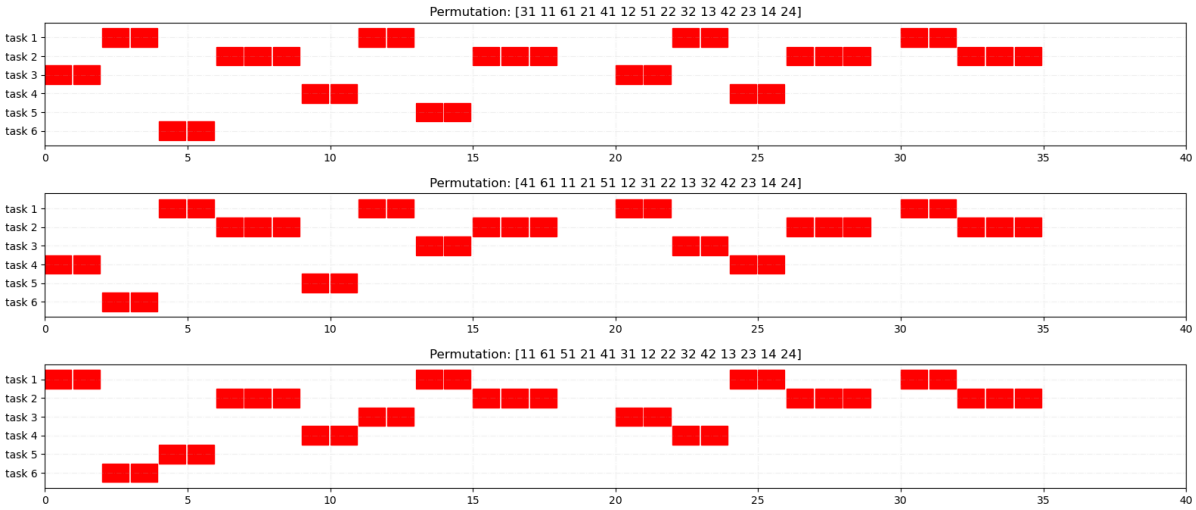
Final assignment : Schedule search



Produced by :

Yoann HOARAU, Salomé MATHE

IPSA - Institut Polytechnique des Sciences Avancées

GitHub project link:
https://github.com/Yawane/IPSA_RTES

Academic year 2024 - 2025

27 April, 2025

Contents

Introduction

Real-time systems are specialized computing environments where correct functionality depends not only on logical accuracy, but also on timely execution. These systems are integrated in applications ranging from industrial automation to medical devices, where meeting deadlines is critical to safety and performance.

A real-time embedded system combines two key characteristics: it is designed for specific tasks and must produce results within deterministic time bounds, called deadlines. We can classify these systems into three categories:

- Hard real-time: a missed deadline cause system failure.
- Soft real-time: a missed deadline degrade quality but don't compromise safety.
- Firm real-time: Late results provide nothing but don't endanger the system.

Objectives

The aim of this report is to prove that our chosen non-preemptive, job-level scheduling algorithm can meet every deadline for the given periodic task set and to construct a concrete schedule that minimizes total waiting time—thereby maximizing processor idle intervals. We will detail our use of the hyperperiod for bounding analysis, compute each job's worst-case response time, and evaluate the algorithm's complexity as tasks scale.

Delays & deadlines

For this assignment, we will focus on non-preemptive scheduling. This type of schedule has a simple mechanism: tasks run uninterrupted until completion. The time tasks will take to complete is given by their corresponding **Worst Case Execution Time (WCET)**. Preemptive schedule has a simplified implementation and predictable execution patterns which are both advantages. However, their is potential deadline misses due to blocking and poor responsiveness to high-priority late-arriving tasks.

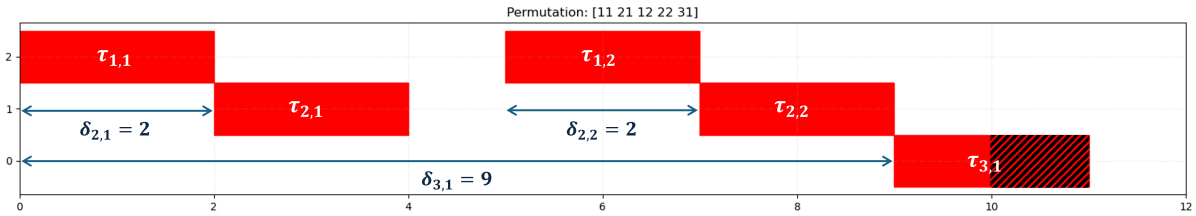


Figure 1: Exeample of schedule with delay and a miss deadline

Task	**WCET**	T_i
τ_1	2	5
τ_2	2	5
τ_3	2	10

Table 1: Simple example task set

You can see in the **Figure 1** above an example of a simple schedule with a three tasks $\{\tau_1, \tau_2, \tau_3\}$ each with a WCET of 2. τ_1 and τ_2 have a deadline of $T_{1/2} = 5$, and for τ_3 the deadline is $T_3 = 10$ (see **Table 1**).

The permutation displayed is as follow $\left[\tau_{11} \ \tau_{21} \ \tau_{12} \ \tau_{22} \ \tau_{31}\right]$. We see that task τ_2 always has a delay of 2 before starting.

However, it always meet deadline, finishing τ_{21} before 5 and τ_{22} before 10. On another hand, task τ_3 never has sufficient time to fully execute. We see it starting with a big delay $\delta_{3,1}$, and finishing after the deadline at 11. If τ_3 had a WCET of 1, it could be executed between time 4 and 5. Actually, this task set is not schedulable, so it would be impossible to find a right way to execute each task without missing a deadline. This was just to illustrate delay and deadline.

From now on, we will use this given task set:

Task	**WCET**	T_i
τ_1	2	10
τ_2	3	10
τ_3	2	20
τ_4	2	20
τ_5	2	40
τ_6	2	40
τ_7	3	80

Table 2: Task set we use for this assignment

1 Schedulability analysis

As shown on the example **Figure 1**, all task set are not schedulable. Before searching for one, it is necessary to compute the total CPU utilisation. It provides guarantee that each task in the system will finish execution within its deadline for the operational life of the embedded system.
For a task τ_i, the CPU Utilization is $U_i = \frac{C_i}{T_i}$, with C_i: the WCET of τ_i. A system is schedulable only if

$$\boxed{U_{Total} = \sum_i U_i < 1}$$

For our system described in **Table 2**, the total CPU Utilization is

$$\begin{aligned} U &= \sum_{i=1}^{7} \frac{C_i}{T_i} \\ &= \frac{2}{10} + \frac{3}{10} + \frac{2}{20} + \frac{2}{20} + \frac{2}{40} + \frac{2}{40} + \frac{3}{80} \\ &= 0.838 < 1 \end{aligned}$$

Therefore, our system is schedulable.

In our algorithm, this verification step is done by a function.

Python code

```python
def is_scheduable(task_set):
    total_utilization = np.sum(task_set[:, 0] / task_set[:, 1]) # C_i / T_i
    print(f"Total Utilization is {total_utilization:<.3f}", end=" ")
    if total_utilization < 1:
        print("--> Is schedulable.")
    else:
        raise ValueError("Task set is not schedulable.")
        # The program stops here.
```

C code

```c
float is_schedulable(int task_set[][2], const int nb_line) {
    float sum = 0.0f;
    for (int i = 0; i < nb_line; i++) {
        sum += (float)task_set[i][0] / (float)task_set[i][1];
    }
    return sum;
    // The main function will decide to stop the program if sum > 1.
}
```

2 Computational complexity

In our implementation, the dominant cost drivers are (1) the worst-case response-time analysis for each job and (2) the construction and evaluation of candidate non-preemptive schedules over the hyperperiod. Let n be the number of task, H their hyperperiod and T_{min} be the smallest period.

The *hyperperiod* is the least common multiple for all task periods. It's the interval after which the job release patterns repeat exactly. In short, if we can organize jobs within a hyperperiod without missing deadlines, the schedule is valid for all time.

1. **Response-time analysis**
   In the worst wase, there are H/T_{min} jobs per task. In the hyperperiod, the total jobs are up to $\mathcal{O}\left(nH/T_{min}\right)$. Then for each job, checking delay and deadline against all the others takes $\mathcal{O}(n)$. Therefore, the total cost of the response-time analysis is

   $$\mathcal{O}\left(n^2 H/T_{min}\right)$$

2. **Non-preemptive schedule search**
   Enumerating all possible permutations of $J = n \cdot H/T_{min}$ is the factorial $J! = J(J-1)(\cdots)$ i.e. $\mathcal{O}\left(J!\right)$ which very quickly becomes immensely high and too high for a normal computer to compute. In fact, our task set has

   $$J = \sum_{i=1}^{7} \frac{80}{T_i} = 29$$

   jobs per permutation, giving a total number of possible permutations of $29! = 8.841 \cdot 10^{30}$. This amount of permutation can't be compute on a low scale time period.
   If we remove τ_7 from the task set, we get $J = \sum_{i=1}^{6} \frac{40}{T_i} = 14$ jobs per permutation, and $14! = 87.1782912 \cdot 10^9$ possible permutations.

3. **Overall complexity**
   Putting both phases together, the algorithm runs in

   $$\mathcal{O}\left(n^2 H/T_{min}\right) + \mathcal{O}\left(J!\right)$$

   For moderate n and small hyperperiods, the overall complexity is acceptable, but will not scale very large task sets or very long hyperperiods.

Because of the explosive complexity, **Section 4** will remove τ_7 from the **Table 2**, and use a hyperperiod of 40 instead of 80.

3 Programming methods

3.1 Job ordering

When we have a task doing multiple jobs until hyperperiod, every job need to be done in order. For example, task 1 τ_1 has 4 jobs: $\{\tau_{1,1}, \tau_{1,2}, \tau_{1,3}, \tau_{1,4}\}$. The permutation array shows in order when a job is computed. A permutation like $\begin{bmatrix}\tau_{1,1} & \tau_{1,3} & \tau_{1,2} & \tau_{1,4}\end{bmatrix}$ is not valid because job $\tau_{1,3}$ is computed before $\tau_{1,2}$.

The function `is_valid` takes a sequence, and verify every job order to return a boolean telling if the sequence is valid or not; job order wise.

Python version

```python
def is_valid(seq):
    n = len(seq)
    for i in range(n):
        task_i, occ_i = divmod(seq[i], 10)
        for j in range(i):
            task_j, occ_j = divmod(seq[j], 10)

            if task_i == task_j and occ_i < occ_j:
                return False
    return True
```

C equivalent

```c
int is_valid(int seq[], const int size) {
    for (int i=0 ; i<size ; i++) {
        int task_i = seq[i] / 10;
        int occ_i = seq[i] % 10;
        for (int j=0 ; j<i ; j++) {
            int task_j = seq[j] / 10;
            int occ_j = seq[j] % 10;
            if (task_i == task_j  && occ_i < occ_j) return 0;
        }
    }
    return 1;
}
```

3.2 Delays & deadlines computation

At this step of the code, we have lots of valid permutations, job wise. Now we need to eliminate all permutations having miss deadlines. For that, we define a function that will put all jobs of all tasks into a timed sequence.

The stragegy here is to keep the permutation array, for example $\begin{bmatrix}\tau_{1,1} & \tau_{1,3} & \tau_{1,2} & \tau_{1,4}\end{bmatrix}$, and create another array with the timing elements, lets call it `timing`. This new array was inspired by the simplest compression method. The timing array is two times as long as the permutation array. For all job with index i in the sequence array, its WCET is store at index $2i$ in the timing array. Into the odd index $(2i + 1)$, there is the amount of pause until the execution of the next start. If we take the simple example on **Figure 1**,

we will have both arrays:

$$\texttt{sequence} = \begin{bmatrix} \tau_{1,1} & \tau_{2,1} & \tau_{1,2} & \tau_{2,2} & \tau_{3,1} \end{bmatrix}$$
$$\texttt{timing} = \begin{bmatrix} 2 & 0 & 2 & 1 & 2 & 0 & 2 & 0 & 2 & 0 \end{bmatrix}$$

Job $\tau_{1,1}$ takes 2 unit of time to execute, then 0 pause because job $\tau_{2,1}$ start right at the end of job $\tau_{1,1}$, with a length of 2 units. Then there is 1 unit of pause because job $\tau_{1,2}$ starts at 5. The process continues. In order to have a timed sequence, we need both arrays.

With that `timing` array, we now know very easily the time at which a job is starting and ending. Job $\tau_{1,2}$ needs to be executed between time 5 and 10. In reality, this job is at index 2 in the `sequence` array, so it starts at time $t_{start} = \sum_{i=0}^{2} \texttt{timing[2i]}$. In is ending at time $t_{end} = t_{start} + \texttt{timing[2i+1]}$.

Using this technique, we wrote these functions calculating delay (difference between optimal start and real start of a job), invalidating the sequence if a job starts too early or miss a deadline.

Python version

```python
def get_timing(seq):
    # ... variables initialization ...
    for c_index in range(len(seq)):
        c = seq[c_index]
        task_i, occ_i = divmod(c, 10)
        normal_start = tab[task_i-1, 1] * (occ_i-1)
        normal_end = normal_start + tab[task_i-1, 1]
        if did_pass[task_i-1]:
            while idx%tab[task_i-1, 1] != 0:
                timing[2*c_index-1] += 1
                idx += 1
                for pos in np.argwhere(idx % tab[:, 1] == 0):
                    did_pass[pos] = False
            did_pass[task_i-1] = False
        did_pass[task_i-1] = True
        for _ in range(tab[task_i-1, 0]):
            timing[2*c_index] += 1
            idx += 1
            for pos in np.argwhere(idx % tab[:, 1] == 0):
                did_pass[pos] = False

        real_start = sum(timing[:2*c_index])
        real_end = real_start + timing[2*c_index]
        if real_start < normal_start or real_end > normal_end:
            return timing, delay, False
        delay += real_start - normal_start
    return timing, delay, True
```

C equivalent

```c
int* get_timing(int tab[][2], int seq[], const int size) {
    // ... Variables initialization ...
    for (int c_index=0 ; c_index<size ; c_index++) {
        int c = seq[c_index];
        int task_i = c / 10 - 1;
        int occ_i = c % 10;
        int normal_start = tab[task_i][1] * (occ_i-1);
        int normal_end = normal_start + tab[task_i][1];
        int count = tab[task_i][0];
        if (did_pass[task_i]) {
            while (idx % tab[task_i][1] != 0) {
                timing[2*c_index-1]++;
                idx++;
                for (int i = 0; i < SIZE; i++) if (idx % tab[i][1] == 0) did_pass[i] = 0;
            }
            did_pass[task_i] = 0;
        }
        did_pass[task_i] = 1;
        for (int i=0 ; i<count ; i++) {
            timing[2*c_index]++;
            idx++;
            for (int i = 0; i < SIZE; i++) if (idx % tab[i][1] == 0) did_pass[i] = 0;
        }
        int real_start = sum(timing, 0, 2*c_index);
        int real_end = real_start + timing[2*c_index];
        if (real_start < normal_start || real_end > normal_end) {
            timing[is_ok_index] = 0;
            return timing;
        }
        timing[delay_index] += real_start - normal_start;
    }
    timing[is_ok_index] = 1;
    return timing;
}
```

3.3 Permutation generation

This will be out main function, generating every possible permutation, and verifying it using previously explained functions as `is_valid()` and `get_timing()`.

Searching for all possible permutations

As said in **Section 2**, the amount of permutation to compute has a cost $\mathcal{O}(J!)$. For our task set, their will be 29! different outcomes to compute, which we will not try because I'm sure to die before the program computes even 10% of all permutations. In order to try the full task set, we need to implement the optimization part; see next **Section 3.4**.

This is the function we made to compute all possible permutations of a given initial sequence:

Python version

```python
def all_permutations(tab):
    # list and variables initialization
    # ...
    while i < n:
        if c[i] < i:
            swap_idx = 0 if i%2 == 0 else c[i]
            temp = a[i]
            a[i] = a[swap_idx]
            a[swap_idx] = temp

            if is_valid(a):
                timing, delay, is_timing_ok = get_timing(a.copy())
                if is_timing_ok:
                    results.append(a.copy())
                    timings.append(timing)
                    delays.append(delay)
            c[i] += 1
            i = 0
        else:
            c[i] = 0
            i += 1
    return np.array(results), np.array(timings), np.array(delays), bad, bad_timings
```

C equivalent

```c
struct permutation* all_permutations(int tab[][2]) {
    // linked list and variable initialization
    // ...
    while (i < length) {
        if (c[i] < i) {
            short int swap_idx = 0;
            if (i%2 != 0) swap_idx = c[i];
            swap(&seq[i], &seq[swap_idx]);

            if (is_valid(seq, length)) {
                int* timing = get_timing(tab, seq, length);
                if (timing[2*length + 1]) {
                    int delay = timing[2*length];
                    int* seq_copy = (int*)malloc(sizeof(int) * length);
                    memcpy(seq_copy, seq, sizeof(int) * length);
                    struct permutation* new = create_perm(seq_copy, timing, delay);
                    append_permutation(&head, new);

                } else free(timing);
            }
            c[i]++;
            i = 0;
        } else c[i++] = 0;
    }
    return head; // head pointer to the linked list of all valid permutations
}
```

3.4 Future optimizations

When generating permutations of a list, efficiency becomes critical. One of the classical algorithms used to generate all permutations is Heap's Algorithm, which produces permutations with minimal data movement and is highly efficient in terms of swaps and recursive calls. However, computing every possible permutation is not always necessary — or even feasible — especially as the number of elements increases, causing factorial growth in the total number of permutations $\mathcal{O}(J!)$.

To address this, we can optimize the permutation process by avoiding the computation of permutations that start with invalid prefixes. This is achieved by integrating a validation check (`is_valid`) during the recursive construction of permutations. The key idea is:

- At each recursive level of permutation generation, consider only a prefix of the permutation.
- Use a validation function (`is_valid(current_prefix)`) to determine whether this prefix could potentially lead to a valid full permutation.
- If `is_valid(current_prefix)` returns False, stop recursion immediately — do not compute or explore any permutations that begin with this prefix.

This is often referred to as backtracking with pruning and can save exponential amounts of time by eliminating large portions of the permutation tree.

With this technique, the complexity of the permutation computation comes to $\mathcal{O}\left(c^J\right)$, with $c > 1 \in \mathbb{R}$. The problem complexity is now exponential, and not factorial.

Optimization implementation using ChatGPT

In our practical experiments on the task set with 6 total tasks, my implementation on the C code required over 6 hours 40 minutes to exhaustively enumerate and evaluate all permutations. Due to time constraints and complexity of the optimizations, I enlisted ChatGPT to help implement the prefix-pruning optimisation quickly, to test it. That same workload completed in roughly 2 hours — a nearly 3× speed-up — demonstrating the power of cutting off invalid branches well before a full permutation is built.

However, even this optimized approach struggles once we move to seven tasks: the factorial growth still overwhelms available time. This underlines that, while pruning dramatically reduces the search space, the underlying complexity remains near factorial. To tackle seven or more tasks, we'll likely need further heuristics such as memorizing partial timing results, applying tighter feasibility bounds to discard subtrees earlier, or even switching to more advanced techniques like branch-and-bound or constraint programming to keep runtimes within practical limits.

4 Results

For the results demonstration, we used the following task set with 6 total tasks:

Task	C	T_i
τ_1	2	10
τ_2	3	10
τ_3	2	20
τ_4	2	20
τ_5	2	40
τ_6	2	40

Table 3: Task set we use for this assignment

After a long night, the console output gives us the results of the scheduling:

main.c - Console Out (modified)

```
1524096 valid permutations.
Best delay is 54
864 equivalent best permutations.

Permutation   1: [ 31 11 61 21 41 12 51 22 32 13 42 23 14 24 ]  delay = 54
Permutation 155: [ 41 61 11 21 51 12 31 22 13 32 42 23 14 24 ]  delay = 54
Permutation 816: [ 11 61 51 21 41 31 12 22 32 42 13 23 14 24 ]  delay = 54
...
Running time: 22777.18500 s
Process finished with exit code 0
```

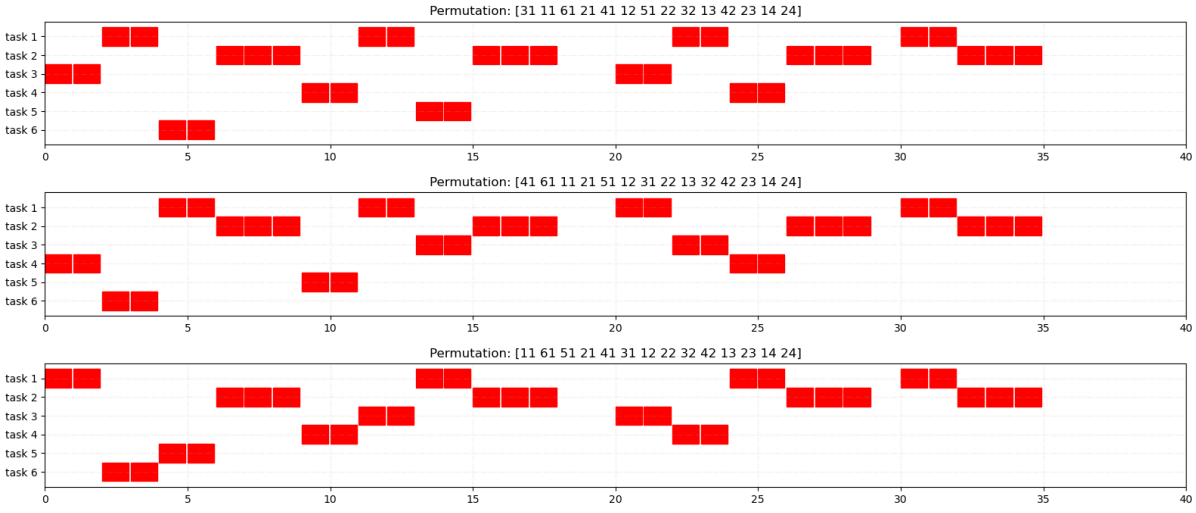


Figure 2: Visual representation of 3 permutations within the best valid ones

In order to get this visual representation, we use a function that takes in argument the sequence and the timing arrays. Here is the Python code I used to do it:

```
seq1 = np.array([31, 11, 61, 21, 41, 12, 51, 22, 32, 13, 42, 23, 14, 24])
seq2 = np.array([41, 61, 11, 21, 51, 12, 31, 22, 13, 32, 42, 23, 14, 24])
seq3 = np.array([11, 61, 51, 21, 41, 31, 12, 22, 32, 42, 13, 23, 14, 24])

timing1, _, _ = get_timing(seq1)
timing2, _, _ = get_timing(seq2)
timing3, _, _ = get_timing(seq3)

show_scheduals(np.array([seq1, seq2, seq3]),
               np.array([timing1, timing2, timing3]),
               n=3)
```

Conclusion

Over the course of this project, we first performed utilization-based schedulability analysis in both Python and C, then implemented exhaustive permutation generation to compute task schedules—only to find that six-task benchmarks required over 6,5 hours hours of runtime.

To address this, one promising optimization is early-prefix pruning, where partial sequences are validated with `is_valid` and entire subtrees are abandoned as soon as a prefix fails, potentially offering a 3× or greater speed-up. I have not implemented this pruning myself due to time constraints, but its theoretical benefits are clear.

Furthermore, by relaxing Task 5's hard deadline — allowing a single miss per hyperperiod — we can reallocate its slack to higher-priority jobs, reducing cumulative delay and further shrinking the search space. Although these optimizations remain to be coded, combining prefix pruning with controlled deadline misses offers a compelling path to practical, low-latency scheduling for seven or more tasks.

Useful links to GitHub and code

GitHub link:
https://github.com/Yawane/IPSA_RTES

These are the elements in the repository:

- `Python_code.py`: The main python code.
- `C_main_code.c`: The main C code, equivalent to the python code. A lot faster.
- `C_OptimizedChatGPT_version.c`: ChatCPT optimized version of my C code.
- `Report_HOARAU_MATHE`: This actual PDF you are reading right now.