Allocation of Real-Time Tasks onto Identical Core Platforms under Deferred fixed Preemption-Point Model

Ikram Senoussaoui
ikram.senoussaoui@univ-lille.fr
CRIStAL, Lille University - LAPECI, Oran1 University
Lille, France

Houssam-Eddine Zahaf
houssam-eddine.zahaf@univ-lille.fr
Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille
F-59000 Lille, France

Mohammed Kamel Benhaoua
k.benhaoua@univ-mascara.dz
LAPECI, Mascara University
Mascara, Algeria

Giuseppe Lipari
giuseppe.lipari@univ-lille.fr
Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille
F-59000 Lille, France

Richard Olejnik
richard.olejnik@univ-lille.fr
Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille
F-59000 Lille, France

ABSTRACT

Deferred-preemption model has been proposed as a compromise between fully preemptive and non-preemptive systems: on one hand they reduce the cache related preemption delays; on the other hand they introduce a small blocking time to higher priority tasks. In this paper, we investigate the problem of allocating a set of real-time tasks with fixed-preemption points onto an identical multi-core platform.

We first propose enumerative and branch-and-bound optimal algorithms, along with techniques to reduce their execution time. Further, we propose a set of heuristics to solve the same problem. We demonstrate the performances of the proposed approaches by the means of a large set of synthetic experiments.

ACM Reference Format:
Ikram Senoussaoui, Houssam-Eddine Zahaf, Mohammed Kamel Benhaoua, Giuseppe Lipari, and Richard Olejnik. 2020. Allocation of Real-Time Tasks onto Identical Core Platforms under Deferred fixed Preemption-Point Model . In *28th International Conference on Real-Time Networks and Systems (RTNS 2020), June 9–10, 2020, Paris, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3394810.3394821

1 INTRODUCTION

The respect of timing constraints of a real-time system requires analyzing the worst case behavior of the real-time software onto the target hardware architecture. It involves a deep knowledge about the interaction between hardware and software to estimate the worst-case execution time (WCET) of all tasks and analyzing their temporal behavior. Several tools have been built to estimate safe upper bounds to the worst-case execution time, e.g. [4], [14]. These bounds are pessimistic when considering preemption as they must include cache-related delays and other run-time costs related to all possible schedules. Cache related delays can be classified into intra-task delays, and inter-task delays. In the first class we account for task cache blocks that can be evicted by the task itself, due to its memory access pattern and to the hardware architecture; these delays can be directly accounted in the WCET estimations. On the other hand, inter-task delays are caused by preemptions. When a preemption occurs, the analysis must account for the time needed to reload cache blocks that have been evicted by preempting tasks. These delays are known as cache-related preemption delays, *CRPD*.

CRPD may be large in fully preemptive systems as a given task can be preempted by any other higher priority task. To reduce the difference between actual and estimated WCET, preemption may be disallowed. However, when a low priority task starts its execution, higher priority tasks are blocked waiting for the low priority task to finish, causing priority inversion.

Limited preemption models have been proposed as an intermediate solution, to overcome the limitations of fully preemptive and non preemptive systems. The *deferred-preemption* model consists in limiting preemption to a set of predefined preemption-points in the task code. This model allows reducing the WCET bounds while still allowing preemption and it is in line with recent hardware accelerators preemption capabilities, such as NVIDIA GPUs. They allow preemption at different costs, according to the type of the task that executes (i.e. *graphical tasks* and *general purpose tasks*). For graphical tasks, the GPU allows preemption at *Draw-Boundaries*, as well as at *Pixel-Level*. When a preemption request is received, the GPU may postpone the preemption to the draw boundaries, or it may stops rasterizing new pixels, complete all operations for the pixels currently in the pipeline, and initiate a context switch. At

RTNS 2020, June 9–10, 2020, Paris, France

ACM ISBN 978-1-4503-7593-1/20/06…$15.00
https://doi.org/10.1145/3394810.3394821

pixel level, the cost of preemption is higher compared to the preemption at draw boundaries, as it requires more state information to save (e.g. register file, etc.). For computational tasks, NVIDIA GPUs allow *Thread-Level* preemption and *Instruction-Level* preemption. The finer the preemption level, the more time consuming preemption is. Therefore, the limited preemption model can offer suitable representation for preemption in GPUs.

In the deferred preemption-point model, the programmer defines a number of preemption points in the task code. During the compilation phase, only a subset of the preemption points is selected, where the task can/may be effectively preempted. The selection process starts by computing the maximum possible time length where a task can be executed without preemption and without violating schedulability. The latter depends on the concurrent tasks and the scheduling policy (FP – Fixed Priority, or EDF – Earliest Deadline Scheduling). Later, independently of the scheduling policy, an algorithm selects the effective preemption points from the list of preemption points defined by the programmer. This technique has been applied only to single-core platforms. In this paper, it is extended to partitioned scheduling in multicore systems.

Contributions . This work present (i) an exact allocation algorithm for a set of real-time tasks presenting fixed-preemption points to identical core platform. We also propose (ii) techniques to reduce the time and space complexity of computing the exact solution as well as (iii) efficient allocation heuristics. Finally, we provide an (iv) exhaustive evaluation of the proposed approaches using a large set of synthetic experiments.

Organization of the paper. The rest of the paper is organized as follows. Section 2 reports related work. Section 3 describes the architecture and the task model. We review the single core preemption-point selection techniques as well as schedulability analysis in Section 4. Our exact allocation algorithm and our proposed heuristics are reported in Section 5 and Section 5.4. Section 6 is reserved to present experimental setup, the results of our simulation and their discussions. We draw conclusions in Section 8.

2 RELATED WORK

Many authors [5, 7–10, 13, 16, 17] have focused on reducing the cost of preemptions in the real-time systems scheduling theory and practice from different perspectives. A good survey overviewing limited preemption models can be found in [11].

In this paper, we are interested in the deferred preemption model (DPM). In such model, a task is divided into a set of non-preemptive chunks separated by preemption-points. At runtime, preemption can only happen at the boundaries of non-preemptive chunks: if a higher priority task arrives while a lower priority task is executing, the preemption is delayed until the next preemption point. DPM increases predictibility and reduces preemption overhead compared to fully preemptive systems, and reduces blocking time compared to non-preemptive scheduling. Two models of preemption handling have been proposed: the floating preemption point model, and the *fixed preemption point model*. In the first model, a preemption point can be inserted at any place of the task code, whereas in the second model the preemption points are fixed prior to analysis.

DPM has been first introduced in [10]. Baruah [5] proposed techniques to compute the maximum length of any interval of time where a given task can execute non-preemptively without violating the schedulability for EDF under the floating preemption points model. The same bounds for fixed priority scheduling has been proposed in [25]. Response time analysis has been improved in [9] by considering special cases where the last non-preemptive chunk can delay the execution of higher priority tasks. Authors in [6] proposed schedulability analysis for both fixed priority and EDF.

In [6], authors tackled the problem of finding the best possible placement of preemption points, they assumed an identical preemption cost for all preemption points. Davis and Bertogna [15] introduced an optimal algorithm for fixed priority scheduling with deferred preemption. Authors in [26] propose schedulability analysis for the fixed preemption model under fixed priority scheduling by considering preemption points with different costs. The latter has been extended in [8] for EDF and using an optimal algorithm to select preemption points while respecting all timing constraints.

Another alternative, called hybrid preemption model, has been presented in [24], based on the Preemption Thresholds Scheduling (PTS) approach in which a task can disable preemption up to a specified priority level(preemption threshold). Each task is assigned a preemption threshold and regular priority also, and it is allowed to preempt only when its priority is higher than the threshold of the preempted task. An exact schedulability analysis for FP with preemption thresholds has been presented in [19].

In order to compute the *Cache Related Preemption Delay* (CRPD), we need to consider different factors: (i) the preemption point *PP* in the code of the preempted task where the preemption occurs, (ii) the cache blocks used until *PP* and that may be reused by the preempted task after preemption, and finally, (iii) the evicting cache blocks of the preempting task [2]. Therefore, the CRPD is bounded by $(R \times BRT)$ where R is an upper bound on the number of reloaded cache blocks and BRT is a cache block reload time.

Among the cache-aware schedulability analyses, Altmeyer et al [3] proposed ECB/UCB-Union Multiset approaches. These approaches account for a more precise number of nested preemptions that can occur during a resource access, comparing to the first exact analyses: ECB/UCB-Union appraoches [2, 23] which consider that the CRPD can be computed by intersecting the *useful cache blocks* and *evicting cache blocks*.

Specific experiments with *Cache Related Preemption Delay* (CRPD) showed that the worst case execution time WCET can increase up to 40% in the presence of preemption when considering a fully preemptive execution [21]. Minimizing cache overhead using limited optimal preemption point placement algorithm using dynamic programming is presented in [12]. The authors have proposed a novel method to calculate the CRPD taking into account the selected preemption points resulting in greater accuracy. Complementary work by Bril et al [1] recently integrated CRPD costs into fixed priority preemption threshold schedulability analysis for taskset with arbitrary deadlines. Optimal priority thresholds are assigned via a CRPD cost minimization.

Both approaches presented in [20, 22] adopt a FP scheduler using deferred preemptions, with the common goal of reducing the preemption overhead by properly placing the preemption points, these

approaches uses heuristic strategy for the placement of preemption points.

All previous researches which are described before consider only single core platforms. When considering multiprocessor scheduling, the allocation problem must be taken into account. According to the moment of taking the allocation decisions and migrations, scheduling algorithms can be classified into partitioned and global. In partitioned scheduling, tasks are allocated to cores and are not allowed to migrate at runtime. Global scheduling algorithms execute at most m (m is the number of available cores) highest priority tasks at the same time. Therefore, tasks are allowed to migrate between cores at runtime. A single job may execute onto more than one core, if it is preempted, leading to higher CRPD.

Authors in [13] studied the schedulability of the DPM under global EDF. In this work, a pseudo-polynomial time schedulability test has been proposed for limited-preemption scheduling under global multiprocessor platforms. In [17] authors propose schedulability analysis for Global Fixed Priority Scheduling with Deferred Preemption for identical cores platforms. They showed that the algorithm introduced in [15] is not optimal in the multiprocessor case.

When considering partitioned scheduling, the schedulability analysis of single core described above can be used to ensure the respect of real-time constraints as a single core level. However, the allocation problem is NP-HARD as it is a particular case of the bin-packing problem. Authors in [17] addressed the allocation problem for partitioned multiprocessor systems using deferred preemption. They used the First-Fit (FF) heuristic to allocate tasks to different processors. Their experimental evaluation showed that partitioned scheduling using deferred preemption provides significantly improved performance over fully preemptive partitioned scheduling and non-preemptive partitioned scheduling.

In this work, we investigate more complex (exact) algorithms that aim at finding an optimal and approximated tasks assignment for partitioned multiprocessor systems using deferred preemption model in a reasonable time. Then, we compare against the classical bin-packing heuristics: First-Fit, Best-Fit and Worst-Fit.

3 SYSTEM MODEL

3.1 Task model

Let $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$ be a set of n tasks. Each task τ_i is a (infinite) sequence of jobs. Each task τ_i is characterized by $(\Gamma_i, \Lambda_i, \mathsf{D}_i, \mathsf{T}_i)$ where :

- $\Gamma_i = \{\gamma_i^1, \ldots, \gamma_i^{\mathsf{np}_i+1}\}$: is the *basic block list*. A basic block $\gamma_i^j \in \Gamma_i$ is a sequential non-preemptive chunk of code of task τ_i. That is, once a basic block starts executing, it can not be preempted by any other higher priority task. Each block γ_i^j is characterized by its worst case execution time denoted by $\mathsf{C}(\gamma_i^j)$. There are $\mathsf{np}_i + 1$ basic blocks in a task, where np_i is the number of preemption points.
- $\Lambda_i = \{\lambda_1, \ldots, \lambda_{np_i}\}$: is the list of preemption points. Preemption point $\lambda \in \Lambda_i$ is the boundary between blocks γ_i^λ and $\gamma_i^{\lambda+1}$. $\mathsf{C}(\tau_i, \lambda)$ is the cost that must be taken into account when preempting task τ_i between basic blocks γ_i^λ and $\gamma_i^{\lambda+1}$.
- D_i: is the task's relative deadline. Each instance of task τ_i must finish its execution no later than D_i time units after its activation.
- T_i: is the task period. We consider sporadic tasks. Therefore, T_i represents the minimum inter-arrival time between the activation of two consecutive jobs of τ_i.

In this paper, we consider constrained deadlines, that is $\mathsf{D}_i \leq \mathsf{T}_i$. Tasks are considered to be independent.

The goal of our work is to select a subset of Λ_i, denoted by $\overline{\Lambda}_i = \{\overline{\lambda}_i^1, \ldots, \overline{\lambda}_i^s\}$, for every task τ_i, such that preemption is allowed only at preemption points in $\overline{\Lambda}_i$, while meeting all deadlines and minimizing preemption costs. Preemption points in $\overline{\Lambda}_i$ are called *effective-preemption points*.

Let $s = |\overline{\Lambda}_i|$ be the number of selected preemption points. The task is then divided into a set of $s + 1$ *non-preemptive regions* $\mathsf{NPR}_i^1, \ldots, \mathsf{NPR}_i^{s+1}$. A non-preemptive region is the union of consecutive non-preemptive blocks between which no preemption point has been selected. They can be expressed as follows:

$$\forall k = 1, \ldots, s \;\; \mathsf{NPR}_i^k = \bigcup_{j=\overline{\lambda}_i^k}^{\overline{\lambda}_i^{k+1}} \gamma_i^j \;\text{ and }\; \mathsf{NPR}_i^{s+1} = \bigcup_{j=\overline{\lambda}_i^k}^{\mathsf{np}_i+1} \gamma_i^j \quad (1)$$

Clearly, for any $\lambda \notin \overline{\Lambda}_i$, $\mathsf{C}(\tau_i, \lambda) = 0$. We define the worst-case execution time of a non-preemptive region to the sum of the execution times of its blocks, plus the preemption cost of a preemption before:

$$C(\mathsf{NPR}_i^k) = \sum_{j=\overline{\lambda}_i^k}^{\overline{\lambda}_i^{k+1}} C(\gamma_i^j) + \mathsf{C}(\tau_i, \overline{\lambda}_i^{k-1}) \quad (2)$$

(without loss of generality we assume that the cost of $\mathsf{C}(\tau_i, \overline{\lambda}_i^0)$ is equal to 0).

Therefore, the total execution time of task τ_i, including the cost of preemption, can expressed as:

$$\mathsf{C}(\tau_i, \overline{\Lambda}_i) = \sum_{j=1}^{\mathsf{np}_i+1} \mathsf{C}(\gamma_i^j) + \sum_{\lambda \in \overline{\Lambda}_i} \mathsf{C}(\tau_i, \lambda) \quad (3)$$

We define by $\mathsf{NPR}_i^{\max}$ the non-preemptive region of τ_i having the largest execution time.

We determine the task utilization as a function of the selected preemption points as follows:

$$u_i(\overline{\Lambda}_i) = \frac{\mathsf{C}(\tau_i, \overline{\Lambda}_i)}{\mathsf{T}_i} \quad (4)$$

Example 3.1. Figure 1 illustrates an example of a task with 3 basic blocks and 2 preemption points. We assume that all preemption points are effective. On the figure, we show the blocks, the non-preemptive regions and the execution times.

3.2 Architecture model

In this paper, we consider a platform composed of m identical cores with partitioned scheduling. Each core has its own single-processor

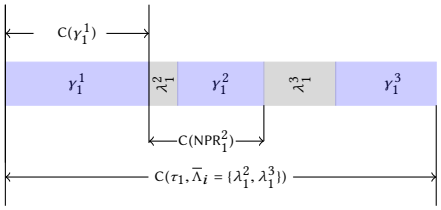

Figure 1: Example of parameters for a task with 3 basic blocks and 2 *effective-preemption points*.

scheduler and a separate ready-queue. Tasks are allocated (partitioned) to the available cores before execution. Compared to global scheduling, partitioned scheduling reduces the overhead due to task migrations. It also simplifies the analysis, because it transforms the scheduling problem on m-identical core platform to an allocation problem and m independent single-processor scheduling problems, for which well-known efficient preemption-point selection techniques exist.

In the rest of the paper, we will first briefly describe the deferred preemption model analysis for single core platforms, and describe further optimal algorithm and heuristics to perform allocation.

4 LIMITED PREEMPTION ANALYSIS FOR SINGLE-PROCESSOR

In this section, we review existing techniques to select preemption points, as well as schedulability analysis for a set of deferred preemption real-time tasks on a single core platform. The schedulability analysis is done in two steps: (i) first, it computes the maximum feasible non-preemptive execution time, (ii) then it selects preemption points using the algorithm proposed in [8].

4.1 Maximum non-preemptive execution-time

Let $Q(\tau_i)$ be the largest non-preemptive interval for task τ_i. It represents the maximum time that task τ_i may execute non-preemptively without violating any timing constraint. Baruah et al. in [5] have proposed techniques to calculate $Q(\tau_i)$ for EDF, similarly in [25] for fixed priority. The algorithm to compute $Q(\tau_i)$ for EDF is reported in Algorithm 1.

Algorithm 1 computing the length of the non-preemptive region

Require: $\mathcal{T}$: Task set
1: deadlines = compute_and_sort_absolute_deadlines($\mathcal{T}$)
2: slack = deadlines[0] − dbf($\mathcal{T}$, deadlines[0])
3: **for** ($\forall d \in$ deadlines − {deadlines[0]}) **do**
4: check_feasibility(d, t^*)
5: slack = min(slack, d − dbf($\mathcal{T}, d$))
6: **if** (slack < 0) **then**
7: **return** false
8: **end if**
9: **if** ($d = D_j$, for some task $\tau_j \in \mathcal{T}$) **then**
10: $Q(\tau_j)$ = slack
11: **end if**
12: **end for**
13: **return** true;

The algorithm starts by listing all deadlines in the interval $[0, t^*]$ (line 1), where t^* is the task set hyper-period, i.e. the least common multiple of all task periods. Then computes the slack as the minimum difference between each deadline and the demand bound function, until the relative deadline of each task. If the slack is negative (Line 6) for some deadline, then the taskset is not schedulable under EDF.

4.2 Selection of Effective Preemption points

At this level, we assume that $Q(\tau_i)$ has been already computed, and the goal of this second step is to select effective preemption-points. Bertogna et al. [8] proposed a sufficient schedulability test (Theorem 4.1): a task set is schedulable if the execution time of any non-preemptive region is less than the maximum non-preemptive interval.

Theorem 4.1 (Bertogna et al. [8]). *A task set $\mathcal{T}$ is schedulable if:*

$$\forall \tau_i \in \mathcal{T}, \quad C(NPR_i^{max}) \leq Q(\tau_i) \tag{5}$$

Several solutions may verify Condition (5). Bertogna's algorithm based on dynamic programming selects a set of effective preemption-points with the goal of optimally reducing the overall preemption overhead [8].

Property 1. We noticed that the optimality of preemption points selection algorithm [8] leads to a lower preemption cost for a more relaxed non-preemptive interval. This property is used to prove Theorem 5.6 in Section 5.

Note that Condition (5) is necessary and sufficient under EDF [8].

5 TASK ALLOCATION

In this section, we present how tasks can be allocated to cores to reduce the overall preemption costs. Let $\mathcal{T}$ be a set of n tasks to allocate on m identical cores. In the rest of this section, we will first present two exact algorithms: (i) an enumerating algorithm able to eliminate branches and solutions either for schedulability or optimality concerns and (ii) an exact algorithm based on Branch and Bound (B&B). Further, we present a set of heuristics to solve the allocation problem.

Our allocation algorithms (exact and heuristic) use a list of not yet-allocated task ordered according to a given criterion. The algorithm selects a task and a core, and attempts to allocate the selected task to the selected core. According to the state of the not yet-allocated task list, the current solution can either be called an *allocation* or a *branch*.

Definition 5.1. Let $\mathcal{T}$ be a set of n tasks to be allocated onto m identical cores. Assignement $\mathcal{S} : \mathcal{T} \rightarrow P \cup \{n_a\}$ is a mapping function defined by:

$$\mathcal{S}(\tau_i) = \begin{cases} p & if\ \tau_i\ is\ allocated\ to\ core\ p \in P \\ n_a & otherwise \end{cases} \tag{6}$$

$\mathcal{S}(\cdot)$ is called an allocation, if: $\forall \tau_i \in \mathcal{T}, \mathcal{S}(\tau_i) \neq n_a$, otherwise, it is called a branch.

We define $Alloc : \mathcal{T} \rightarrow \{\mathcal{S}_1(\cdot) \ldots \mathcal{S}_x(\cdot)\}$, as the set of x possible allocations, where x is a finite number of allocations.

Definition 5.2. Let $\mathcal{S}_k(\cdot)$ be an allocation (resp. a branch). We denote by $\text{cost}(\mathcal{S}_k(\cdot))$ the preemption cost of allocation (resp. branch) $\mathcal{S}_k(\cdot)$. It can be computed as follows:

$$\text{cost}(\mathcal{S}_k(\cdot)) = \sum_{j=1}^{m} \sum_{\tau_i \in \mathcal{T}_j} \sum_{\bar{\lambda}_i^l \in \bar{\Lambda}_i} \frac{\mathsf{C}(\bar{\lambda}_i^l)}{\mathsf{T}_i} \tag{7}$$

5.1 Enumerating algorithm

The exact enumerating algorithm explores the space solution by solution. It is able to either cut a branch or to eliminate an allocation and preserve the optimal solution. An overview of our enumerating procedure is disclosed in Algorithm 2.

Algorithm 2 generate_evaluate_solutions($\mathcal{T}, \mathcal{P}, \mathcal{S}_{\text{curr}}$)

```
1:  if (T = ∅) then
2:    if (cost(S_curr) < cost(S_best)) then
3:      S_best = S_curr
4:      return
5:    end if
6:  end if
7:  τ_i = select the shortest relative deadline task from T
8:  for ( p ∈ P ) do
9:    allocate τ_i to p for the current allocation S_curr
10:   if (schedulable(S_curr, τ_i, p)) then
11:     S_new = S_curr
12:     generate_evaluate_solutions(T \ τ_i, P, S_new)
13:   end if
14: end for
```

The algorithm generates all allocations recursively, it takes as input the set of all tasks in $\mathcal{T}$ sorted by relative deadlines in increasing order, the set of cores $\mathcal{P}$ and the current branch/allocation, and it returns the solution with the minimum preemption cost $\mathcal{S}_{best}$. In the beginning, it selects a task and a processor and it tries to allocate the selected task to the selected processor (Line 9). Therefore, it tests the schedulability (Line 10) for the concerned core, using Algorithm 3. Further, the algorithm removes the studied task from $\mathcal{T}$, and executes the recursive call (Line 12) for the new branch/allocation $\mathcal{S}_{new}$. If this test fails then, the branch/allocation is aborted. Once $\mathcal{T}$ is empty, the algorithm saves the best solution and continues to evaluate the next one. The algorithms repeats the latter operations for every task on all cores.

Algorithm schedulable (Algorithm 3), which tests the schedulability of the selected core p, uses properties of algorithm 1 and Property 1.

It takes as input the selected task, the concerned core p and the task set already allocated to p. First, it tests the schedulability using a fast necessary utilisation based-test (Line 1). If the test is successful, then it computes the maximum of the *non-preemptive interval* of tasks already allocated to p using Algorithm 1 (Line 2). The latter checks the schedulability until the hyper-period using dbf-based test (lines 5-8 in Algorithm 1). When schedulability test of algorithm 1 is performed then, our algorithm selects the effective preemption points by invoking the algorithm presented in [8] to determine the *non-preemptive regions* only for the studied task τ_i (Line 4). If condition (5) is respected during all the effective preemption points selection process, then, the algorithm updates the total execution time of the task τ_i, including the cost of preemption (Lines 6 and 7). If these tests fail, then the algorithm aborts on fail.

Therefore, when adding a new task to the current branch, the already computed maximum non-preemptive region and the selected points for the already allocated tasks do not need to be recomputed.

Algorithm 3 schedulable($\mathcal{S}_{\text{curr}}, \tau_{\text{i}}, \text{p}$)

```
1:  if (fast_utilization_test(p)) then
2:    res_Q = compute_max_length_NPR(S_curr) using Algo 1
3:    if (res_Q) then
4:      res_npr = compute_NPR(τ_i, Q(τ_i)) using Algo in [8]
5:      if (res_npr) then
6:        update_C(τ_i)
7:        update_total_cost(p)
8:        return true
9:      end if
10:   end if
11: end if
12: return false
```

5.1.1 *Optimality of the enumeration algorithm.*

To prove the optimality of the enumeration algorithm, we prove Theorem 5.3 below.

Theorem 5.3. *The enumeration algorithm explores all schedulable allocations and selects the optimal one.*

Proof. The proof derives from the recursive structure of the algorithm. For every task, all possible cores are tried (Line 8), and the algorithm invokes itself every time with a smaller set $\mathcal{T}$ (Line 12). Moreover, all schedulable solutions are explored at (Line 1), thus the best solution is selected. □

5.2 Branch and Bound

In this section, we present a recursive algorithm based on branch and bound (Algorithm 4), and prove its correctness.

The algorithm uses a set of non allocated tasks $\mathcal{T}$ in the current branch/allocation $\mathcal{S}_{curr}$ and a list $\mathcal{L}$ of not-yet-finished branches. At each iteration, it selects a branch and tries to allocate the shortest relative deadline task in $\mathcal{T}$ (selected in Line 8) to every core in the platform. Thus, it creates m new branches at each iteration.

For each new branch, we test schedulability : (i) without a complete evaluation of the branch (allocation) using Theorem 5.6 and Lemma 5.7 detailed below (Line 10); (ii) If not possible, it evaluates the maximum non-preemptive region length and selects the *effective-preemption points* only for the selected task (Line 11) using dbf-based test according to Algorithm 3. If these tests fail, then the branch is discarded. In the opposite case, the branch is added to the list of not-yet-finished branches $\mathcal{L}$ (Line 12). Once, the non-yet-allocated task list $\mathcal{T}$ for the current branch is empty, the algorithm compares the allocation to the best known solution. If it improves it, the solution is saved and the lower-bound is updated (Lines 3-5). The latter is used to eliminate all branches having a preemption cost greater than the new lower-bound (Line 17).

While the not-yet-finished branches $\mathcal{L}$ is not empty, the algorithm selects a new branch to be explored in the next iteration (Line 21) according to two alternative criteria:

- the branch having the least preemption cost is selected first (depth-first);
- the branch having the least number of tasks in its non-yet-allocated list, is selected first (breadth-first).

Algorithm 4 branch_and_bound(S_{curr}, bound)

Require: global variables: $\mathcal{L} = \emptyset$, S_{best}
1: $\mathcal{T}$ = set of non allocated tasks in S_{curr}
2: **if** ($\mathcal{T}$ == $\emptyset$) **then**
3: **if** (cost($\mathcal{S}_{curr}$) < bound) **then**
4: $S_{best} = S_{curr}$
5: bound = cost(S_{curr})
6: **end if**
7: **else**
8: τ = select the shortest relative deadline task from $\mathcal{T}$
9: **for** ($\forall p \in \mathcal{P}$) **do**
10: **if** (not_dominated(S_{curr}, S_{best})) **then**
11: **if** schedulable(S_{curr}, τ, p) **then**
12: $\mathcal{L} = \mathcal{L} \cup \{S_{curr}[\tau$ allocated on $p]\}$
13: **end if**
14: **end if**
15: **end for**
16: **end if**
17: $\mathcal{L}$ = eliminate_branches($\mathcal{L}$, bound)
18: **if** ($\mathcal{L}$ == $\emptyset$) **then**
19: **return** S_{best};
20: **end if**
21: S = select_the_minimum_branch($\mathcal{L}$)
22: branch_and_bound(S, bound);

In the rest of this section, we will show how schedulability can be tested without a complete evaluation (i.e. without invoking the preemption-points selection process).

Definition 5.4. Let $\mathcal{S}_i(\cdot)$ and $\mathcal{S}_j(\cdot)$ be two distinct allocations, (resp. branches) ($i \neq j$).

We define the relation order > as follows:

$$\mathcal{S}_i(\cdot) > \mathcal{S}_j(\cdot) \implies \text{cost}(\mathcal{S}_i(\cdot)) < \text{cost}(\mathcal{S}_j(\cdot)) \tag{8}$$

Relation > orders allocations (resp. branches) according to their preemption costs. Note that in the case of equal costs, we can not judge if $\mathcal{S}_i(\cdot)$ is better than $\mathcal{S}_j(\cdot)$.

Definition 5.5. Let $\mathcal{S}_1(\cdot)$ and $\mathcal{S}_2(\cdot)$ be two distinct allocations. We denote by τ_i a task in allocation $\mathcal{S}_1(\cdot)$, and by τ_i' the same task in allocation $\mathcal{S}_2(\cdot)$.

We define the relation order $\gg$ as follows :

$$\mathcal{S}_1(\cdot) \gg \mathcal{S}_2(\cdot) \implies \forall i, \mathrm{Q}(\tau_i) \geq \mathrm{Q}(\tau_i') \tag{9}$$

The relation order $\gg$ allows to define a dominance relation between two allocations by calculating only maximum non-preemptive regions lengths $\mathrm{Q}(\tau_i)$, rather than a complete selection of preemption points.

Theorem 5.6. *Let $\mathcal{S}_1(\cdot)$ and $\mathcal{S}_2(\cdot)$ be two distinct allocations. $\mathcal{S}_1(\cdot) \gg \mathcal{S}_2(\cdot) \implies \mathcal{S}_1(\cdot) > \mathcal{S}_2(\cdot)$*

Proof. The proof is derived from Property 1 and Definitions 5.4, 5.5. Let consider $\mathrm{Q}(\cdot)$ be the maximum non-preemptive region for a given task on allocation $\mathcal{S}_1(\cdot)$ and $\mathrm{Q}(\cdot)'$ be the maximum non-preemptive region for the same task on allocation $\mathcal{S}_2(\cdot)$. We assume $\mathrm{Q}(\cdot) > \mathrm{Q}(\cdot)'$. From Property 1, it follows that the preemption cost in allocation $\mathcal{S}_1(\cdot)$ is less than the preemption cost in allocation $\mathcal{S}_2(\cdot)$. According to Definition 5.4 $\mathcal{S}_1(\cdot) > \mathcal{S}_2(\cdot)$, proving the theorem. □

According to Theorem 5.6, it is not necessary to compute the preemption points to test domination between two allocations. In fact, we can avoid computing the preemption points for the dominated solution, further reducing the execution time of the algorithm.

Lemma 5.7. *Let consider n tasks to allocate to m processors, with: $n > m$*

Any feasible solution $\mathcal{S}(\cdot)$, having at least one free processor (without any task), is dominated.

Proof. The proof is straightforward from Theorem 5.6. In fact, having a free processor implies that for any non-empty processor, a task can be selected and reallocated to the free processor, thus producing a higher maximum-non-preemptive region length. Therefore, the new solution dominates the one with an empty processor according to Theorem 5.6. □

According to this lemma, it is not necessary to evaluate the allocations having at least an empty processor: the algorithm needs not to compute the maximum non-preemptive region length as it will not lead to optimal solution.

The theorems and lemmas described in this section are used to eliminate allocations at (Lines: 10 and 17) in Algorithm 4.

5.2.1 *Branch and bound optimality proof.*

First, we prove that all branches generated by the branch and bound algorithm having a preemption cost greater than the lower bound are dominated by the best known solution. Further, we demonstrate that our algorithm preserves the optimal solution at each branch level.

Lemma 5.8. *Let $S(\cdot)$ be a branch in the set of not-yet-finished branches $\mathcal{L}$, and let* cost($\mathcal{S}(\cdot)$) > *bound. Then all solutions belonging to the branch $S(\cdot)$ have larger cost than the currect bound.*

Proof. When new branches are explored, tasks can only be added to cores. Since tasks are sorted by deadlines in non-decreasing order, the preemption cost can only increase. Thus, the branch $S(\cdot)$ having a preemption cost greater than the lower bound, can not lead to a better solution than the best-known. □

Let us demonstrate now that all branches discarded using our algorithm are not optimal.

Theorem 5.9. *Function eliminate_branch never eliminates the optimal solution.*

Proof. From the previous lemma, eliminate_branch eliminates all solutions with cost greater than the lower bound, and since we have already found a solution S_{best} whose cost is equal to the lower bound, then it follows that eliminate_branch cannot eliminate the optimal solution. □

To better clarify how the proposed branch and bound algorithm works, we propose an example in the following section.

Example 5.10. Let $\mathcal{T}$ be the task set to be allocated onto 2 cores. Tasks characteristics are described in Table 1. In Figure 2, we report a sub-tree of the branches eveluated by our branch and bound algorithm. Each node in the figure describes a branch. It contains the branch identifier, its cost and the already allocated tasks for the two cores.

task	D_i	T_i	Γ_i	Λ_i
τ_1	1413	1500	{212,171,344,66,249}	{ 0,46,78,14,47 }
τ_2	5673	6000	{17,54,490,101,418,74}	{0,14,94,21,74,13}
τ_3	1498	1500	{146,347,136,37,121}	{0,90,32,7,19}
τ_4	1277	1500	{17,31,43,3,24,6}	{ 0,6,8,0,3,0 }

Table 1: Task parameters

Before starting, the algorithm sorts the tasks by increasing relative deadline. It then starts by selecting the first task τ_4, because it has the least deadline 1277. Thus two branches are created (2 and 3 in Figure 2), each of cost equal to 0. For the 2^{nd} iteration, task τ_1 is selected. As branch 2 and branch 3 have the same cost (equal to 0), we select arbitrarily branch 2. Task τ_1 can be either allocated along with τ_4 onto the same core as shown in branch 5. It can also be allocated to the other core, therefore alone on the empty processor. In both branches, the preemption cost is equal to 0, therefore we select branch 5 as it leaves an empty core, therefore giving us more chances to lead to an optimal solution. The next task to allocate is task τ_3. The algorithm has the two possibilities: (i) allocate τ_3 together with τ_4 and τ_1 on the same core or (ii) allocate τ_3 alone on the empty core. In the first solution, schedulability fails, therefore without further exploring, the branch it is eliminated. In the second, the schedulability cost is equal to 0 as the task is allocated on an empty processor. Further, the algorithm selects branch 7. The next task to evaluate is task τ_2. It can be either allocated with τ_4 and τ_1 on the same core, having a preemption cost of 0.008 (Allocation 9) or with τ_3 onto its core, having a preemption cost of 0.0058 (Allocation 8). The not yet allocated task list is now empty, therefore the bound is updated to 0.0058 and the allocation is saved as the best known. All branches having preemption costs greater than 0.0058 are eliminated.

5.3 Computational Complexity

Regarding to run-time complexity, the proposed exact algorithms evaluate the maximum non-preemptive region length and selects the preemption points for the selected task at every tree level except the root, yielding to a time complexity of $O(n! \times m)$ in the worst case. However, the run-time of the branch-and-bound algorithm in the average case is likely less.

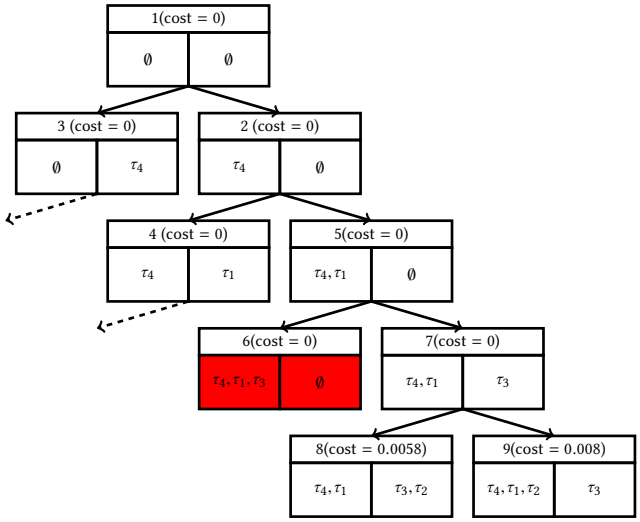

Figure 2: Example of branch and bound

5.4 Allocation heuristics

Task allocation problem is known to be NP-hard in the strong sense. Thus, even with the proposed optimizations and dynamic programming, the complexity of finding an optimal solution is high. Therefore, we use classical bin packing heuristics as an alternative allocation.

Algorithm 5 Heuristics($\mathcal{T}$, p, alloc[FF, BF, WF], ORDER)

```
1:  sort_tasks(ORDER)
2:  for (∀τ ∈ T) do
3:      allocated = false
4:      sort processors for BF and WF
5:      for (∀p ∈ P) do
6:          S_curr = select tasks on p ∪ {τ}
7:          if (total_schedulable(S_curr, τ, p)) then
8:              allocated = true
9:              allocate task into core p
10:             break;
11:         end if
12:     end for
13:     if (allocated = false) then
14:         No task allocation is found
15:     end if
16: end for
17: Return tasks allocation
```

In practice, First-Fit (FF), Best-Fit (BF) and Worst-Fit (WF) operate in a similar fashion. First, tasks are sorted before the allocation according to their deadline, density, or laxity (Line 1). Then, the algorithm selects the tasks on the top of the order relation. Unlike the algorithm FF, BF and WF sort the cores by capacity. For BF the cores are sorted in a decreasing order of their utilizations, whereas in the case of WF, they are sorted in increasing order of utilization. The heuristic tries to allocate the selected tasks to the first processor. If the allocation fails (for schedulability), the next

processor is investigated. When all processors are investigated and none of the allocations have been found feasible, the heuristics aborts on fail. The algorithm 5 describes FF, BF and WF heuristics. The schedulability is checked at the same time when calculating maximum-non-preemptive region length and when selecting preemption points using Algorithm "*total_schedulable*" .

When tasks are sorted according to their deadline, at each allocation, total_schedulable algorithm is similar to Algorithm 3, as it is not necessary to recompute the maximum-non-preemptive region length and preemption points for already allocated tasks, However, total_schedulable algorithm recomputes it for all other sorting mechanisms as they do not ensure that only tasks with shorter deadlines have been allocated before.

6 RESULTS AND DISCUSSIONS

In this section, we evaluate the performance of our schedulability analysis and allocation strategies. We compare our optimal algorithms against classical allocation heuristics: First Fit (FF), Best Fit (BF) and Worst Fit (WF).

Due to the complexity of computing the exact solution, and for sake of fast evaluation, we consider a hardware platform compound of 3 identical processing units, and a large set of synthetic task sets, each comprising 24 tasks.

6.1 Task Generation

We apply our heuristics on a large number of randomly generated synthetic task sets.

The task set generation process takes as input n (the number of tasks) and U (the target total utilization). First, we start by generating the utilization of the n tasks by using the UUniFast-Discard [18] algorithm. Further, for every utilization u_i, we generate randomly the number of basic blocks k between 8 and 15. We generate block utilization by generating randomly k utilizations using UUniFast algorithm with total utilization equal to the task utilization.

To avoid intractable hyper-periods, the period of every task is generated randomly according to values taken from a list where the minimum is 120 and the maximum is 120,000 by step of 500.

Further, we inflate each block utilization by the task period to generate the block execution time. Then, we generate the block preemption cost by generating a random value P between 0.1 and 0.2. P is the percentage of the block utilization that represents the block preemption cost. The first block preemption cost is equal to 0. The task deadline is generated randomly between $[0.75 \cdot T_i, T_i]$.

6.2 Simulation results and discussions

We varied the baseline utilization from 0.25 to 3.75 with a step of 0.25. In all the graphs presented in this section, each point is the average value of 100 executions. We reports the results of the schedulability and the timing complexity of optimal solutions and heuristics.

In Figure 3 we report the results of the schedulability of the optimal allocation algorithms (OPT) against the best fit (BF-DD), worst fit (WF-DD) and first fit (FF-DD) algorithms as a function of total utilization. For all the algorithms, the input tasks are sorted according to their relative deadlines in non-decreasing order. At low total utilization values, all algorithms are able to schedule all tasks sets.

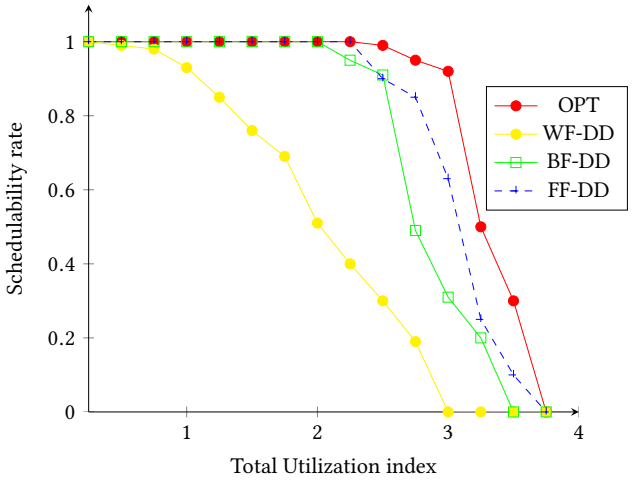

Figure 3: Schedulability for optimal solutions against BF, WF and FF

It is clear that any version of the optimal algorithm dominates the heuristics. As utilization increases, FF-DD and BF-DD algorithms outperform the WF-DD algorithm, the latter schedulability falls drastically as utilization increases.

In fact, the allocation process depends on the already allocated tasks. As in WF the tasks are allocated according to the worst case utilization, tight deadline tasks may be allocated along with large deadline tasks. Therefore, the latter do not have enough slack to be executed non-preemptively, causing schedulability failure. In contrast, when using best fit or first fit algorithms, tight deadlines tasks are allocated together, allowing to have the same tight slacks, but as deadlines are closer, their execution requirements are closer (deadlines are generated as 0.75 of the task period, based on which task execution time is generated). Although we have only three cores, the optimal algorithm is able to achieve high schedulability rates even greater than the maximum number of cores.

To explain this fact, we remark that the preemption cost is inflated from the task execution time. Therefore the maximum schedulable utilization, when selecting all preemption points is equal to the number of cores, however when not selecting all preemption points, the actual utilization is less than the maximum theoretical generated utilization, which is used in the plots.

We show in Figure 4 a comparison between optimal solutions and heuristics as a function of the required time to complete the analysis. The required time for analysis using the enumerative algorithm is very large compared to those of the others algorithms. The optimal branch-and-bound algorithms require more time than heuristics, as expected. In fact, at any scenario, the enumerative algorithm will explore all the design space, thus it is very time-consuming, however the two branch-and-bound implementations may cut a branch without evaluation, therefore they are faster. The heuristic algorithms explore only a subset of the design space, and hence are the fastest. Algorithm OPT-P1 denotes the branch and bound algorithm where the next explored branch is the branch having the least cost, while in OPT-P2 the branch having the least size of the not-yet allocated tasks is selected first. Algorithm OPT-P1 has

a lower average complexity compared to OPT-P2. In fact, OPT-P2 tries to find a low bound faster than OPT-P1, therefore has a better capacity to cut branches.

As the enumerative algorithm has a very high average complexity, the number of tasks in this setup has been limited to 8 tasks and the number of processors to 3 to be able to achieve a large number of simulations.

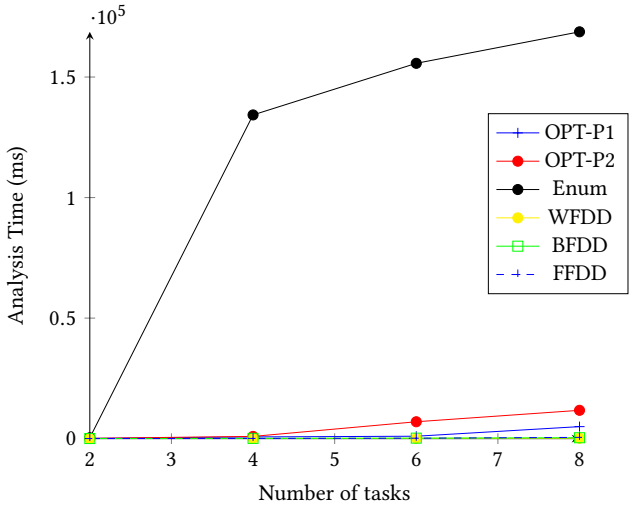

Figure 4: The analysis time as a function of number of tasks

In Figure 5, we evaluate the impact of the task set size on the effectiveness of the analysis to determine the task set schedulability. In this setup, the task set utilization was fixed at $U_{\mathcal{T}} = 3$, and the evaluated task set size ranges from 4 to 24. We notice that the schedulability rates is higher for task set with large number of tasks. In fact, tasks present more *effective-preemption points* in large task set, therefore the total preemption cost of each task increase allowing the execution time to be reduced, hence, schedulability increase as shown in Figure 5.

In Figure 6 and Figure 7 we focus on the impact of the sorting algorithms on FF, BF and WF heuristics.

Figure 6 reports schedulability as a function of total utilization when input tasks are sorted in increasing (respectively descreasing) order of task density. Please, notice that sorting tasks in decreasing order of density allows to achieve higher schedulability rates compared to the increasing density order. In fact, when using increasing order, FF and BF tend to allocate the small tasks, having a small density on the same core, and the more heavy tasks to be allocated in a small number of cores, thus reducing schedulability. WF presents the opposite behavior, but it also leaves the heaviest tasks to be allocated at the end, thus leading to schedulability failure. When ordering by decreasing order, BF, WF and FF performances have similar behavior as they start by allocating heavy tasks first, and further smaller tasks are inserted on the cores where they can be schedulable.

Figure 7 reports schedulability as a function of total utilization when input tasks are sorted in increasing (respectively descreasing) order of task laxity.

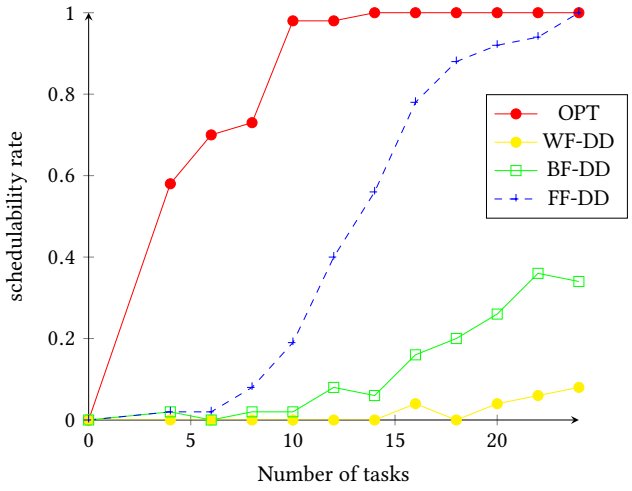

Figure 5: Schedulability at different taskset size.

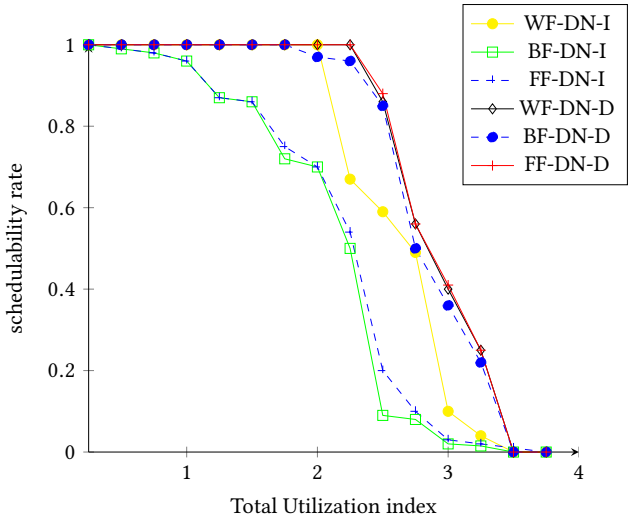

Figure 6: Performance of BF, WF and FF using sorted tasks by density

When ordering by increasing order, FF and BF are equal, the same performance behavior is noticed for BF and WF when task are sorted by decreasing order of laxity. Noticed that sorting tasks by laxity allows to have slightly better performances compared to density sorting.

7 ACKNOWLEDGEMENT

This research has been funded in part By PHC-CURIEN TASSILI 19MDU213 grant, PRIMA WATERMED 4.0

8 CONCLUSION AND FUTURE WORK

In this paper, we presented allocation algorithms for real-time tasks with fixed preemption points on an identical core platform. We have proposed two optimal algorithms and a set of heuristics to

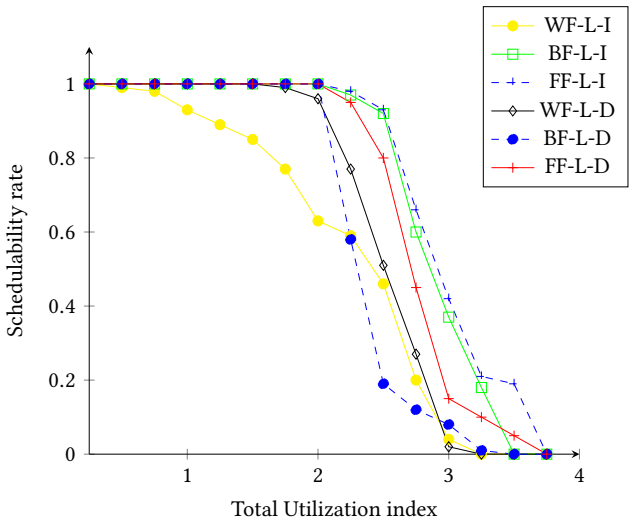

Figure 7: Performance of BF, WF and FF using sorted tasks by laxity

effectively achieve allocation while meeting all deadlines, and minimizing the preemption costs.

We have presented the performances of the proposed approaches using a large set of synthetic expirements. The branch and bound implementations have shown a good compromise between computational time and the quality of produced solution.

We plan to extend the proposed approaches to consider heterogeneous platforms, as those found in recent NVIDIA embedded boards, therefore considering dependent tasks on heterogeneous platforms.

REFERENCES

[1] Integrating cache-related pre-emption delays into analysis of fixed priority scheduling with pre-emption thresholds. In *Proceedings Real-Time Systems Symposium (RTSS 2014)*, pages 161–172. IEEE, 12 2014.

[2] Sebastian Altmeyer and Claire Maiza Burguière. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture*, 57(7):707–719, 2011.

[3] Sebastian Altmeyer, Robert I Davis, and Claire Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5):499–526, 2012.

[4] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa: an open toolbox for adaptive wcet analysis. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, pages 35–46. Springer, 2010.

[5] Sanjoy Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 137–144. IEEE, 2005.

[6] Marko Bertogna, Giorgio Buttazzo, Mauro Marinoni, Gang Yao, Francesco Esposito, and Marco Caccamo. Preemption points placement for sporadic task sets. In *2010 22nd Euromicro Conference on Real-Time Systems*, pages 251–260. IEEE, 2010.

[7] Marko Bertogna and Michele Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 149–160. IEEE, 2007.

[8] Marko Bertogna, Orges Xhani, Mauro Marinoni, Francesco Esposito, and Giorgio Buttazo. Optimal selection of preemption points to minimize preemption overhead. In *23rd Euromicro Conference on Real-Time Systems.*, pages 217–227, 2011.

[9] Reinder J Bril, Johan J Lukkien, and Wim FJ Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *Real-Time Systems*, 42(1-3):63–119, 2009.

[10] Alan Burns. *Preemptive priority based scheduling: An appropriate engineering approach.* University of York, Department of Computer Science, 1993.

[11] Giorgio C Buttazzo, Marko Bertogna, and Gang Yao. Limited preemptive scheduling for real-time systems. a survey. *IEEE Transactions on Industrial Informatics*, 9(1):3–15, 2012.

[12] John Cavicchio, Corey Tessler, and Nathan Fisher. Minimizing cache overhead via loaded cache blocks and preemption placement. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 163–173. IEEE, 2015.

[13] Bipasa Chattopadhyay and Sanjoy Baruah. Limited-preemption scheduling on multiprocessors. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, page 225. ACM, 2014.

[14] Antoine Colin and Isabelle Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Proceedings 13th Euromicro Conference on Real-Time Systems*, pages 37–44. IEEE, 2001.

[15] Robert I Davis and Marko Bertogna. Optimal fixed priority scheduling with deferred pre-emption. In *2012 IEEE 33rd Real-Time Systems Symposium*, pages 39–50. IEEE, 2012.

[16] Robert I Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys (CSUR)*, 43(4):35, 2011.

[17] Robert I. Davis, Alan Burns, Jose Marinho, Vincent Nelis, Stefan M. Petters, and Marko Bertogna. Global and partitioned multiprocessor fixed priority scheduling with deferred preemption. *ACM Trans. Embed. Comput. Syst.*, 14(3):47:1–47:28, April 2015.

[18] Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *WATERS*, 2010.

[19] Uğur Keskin, Reinder J Bril, and Johan J Lukkien. Exact response-time analysis for fixed-priority preemption-threshold scheduling. In *2010 IEEE 15th Conference on Emerging Technologies & Factory Automation (ETFA 2010)*, pages 1–4. IEEE, 2010.

[20] Sheayun Lee, Chang-Gun Lee, Minsuk Lee, Sang Lyul Min, and Chong Sang Kim. Limited preemptible scheduling to embrace cache memory in real-time systems. In *Languages, Compilers, and Tools for Embedded Systems*, pages 51–64. Springer, 1998.

[21] Rodolfo Pellizzoni, Bach D Bui, Marco Caccamo, and Lui Sha. Coscheduling of cpu and i/o transactions in cots-based embedded systems. In *2008 Real-Time Systems Symposium*, pages 221–231. IEEE, 2008.

[22] Jonathan Simonson and Janak H Patel. Use of preferred preemption points in cache-based real-time systems. In *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*, pages 316–325. IEEE, 1995.

[23] Yudong Tan and Vincent Mooney. Timing analysis for preemptive multitasking real-time systems with caches. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(1):7–es, 2007.

[24] Yun Wang and Manas Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No. PR00306)*, pages 328–335. IEEE, 1999.

[25] Gang Yao, Giorgio Buttazo, and Bertogna Marko. Bounding the maximum length of non-preemptive regions under fixed priority. In *2009 15th IEEE Internationnal Conference on Embedded and Real-Time Computing Systems and Applications*, pages 351–360. IEEE, 2009.

[26] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. Feasibility analysis under fixed priority scheduling with limited preemptions. *Real-Time Systems*, 47(3):198–223, 2011.