2017 IEEE 20th International Symposium on Real-Time Distributed Computing

An Analysis of Lazy and Eager Limited Preemption Approaches under DAG-based Global Fixed Priority Scheduling

Maria A. Serrano[*][†], Alessandra Melani[‡], Sebastian Kehr[§], Marko Bertogna[¶], Eduardo Quiñones[*]

[*]Barcelona Supercomputing Center (BSC), Barcelona, Spain
[†]Universitat Politecnica de Catalunya (UPC), Barcelona, Spain
[‡]Scuola Superiore Sant'Anna, Pisa, Italy
[§]DENSO AUTOMOTIVE Deutschland GmbH
[¶]University of Modena and Reggio Emilia, Modena, Italy

***Abstract*—DAG-based scheduling models have been shown to effectively express the parallel execution of current many-core heterogeneous architectures. However, their applicability to real-time settings is limited by the difficulties to find tight estimations of the worst-case timing parameters of tasks that may arbitrarily be preempted/migrated at any instruction. An efficient approach to increase the system predictability is to limit task preemptions to a set of pre-defined points. This limited preemption model supports two different preemption approaches, *eager* and *lazy*, which have been analyzed only for sequential task-sets.**

This paper proposes a new response time analysis that computes an upper bound on the lower priority blocking that each task may incur with eager and lazy preemptions. We evaluate our analysis with both, synthetic DAG-based task-sets and a real case-study from the automotive domain. Results from the analysis demonstrate that, despite the eager approach generates a higher number of priority inversions, the blocking impact is generally smaller than in the lazy approach, leading to a better schedulability performance.

I. Introduction

The use of multi- and many-core embedded architectures in real-time systems is motivated by the demand for increased computational performance [1][2]. The introduction of these architectures involves a twofold challenge: on one side, arguments about system correctness with guaranteed processing response times must be provided; on the other side, a convincing timing analysis must be coupled with the adoption of parallel programming models. The use of parallel programming models is of paramount importance to efficiently exploit the huge performance opportunities of many-core architectures.

This paper analyses the global fixed priority scheduling problem of real-time task-sets composed of DAG tasks [3], where each task is divided into multiple smaller computation units, called sub-tasks, that are allowed to execute simultaneously on different cores, provided the required precedence constraints are met. Interestingly, this scheduling model has certain similarities with the OpenMP tasking model [4], the de-facto standard for shared memory parallel programming in high-performance computing (HPC), and supported by the newest many-core embedded architectures.

A key aspect in real-time scheduling is the preemption strategy used as it allows the operating system to allocate tasks requiring urgent service. Traditionally, real-time systems relied on fully preemptive (FP) or fully non-preemptive (FNP) scheduling strategies [5]. The former specifies that a task can be preempted at any point during its execution if a higher priority task arrives. The latter does not allow tasks to be preempted, being always executed until completion. Despite there is no blocking time from lower priority tasks in FP systems, they may incur important overheads due to context switch costs, cache related preemption and migration delays, and increased resource contention costs [6][7], significantly increasing the pessimism in the worst-case execution times. Moreover, a FP system may produce unnecessary preemptions leading to unnecessary, and possibly prohibitively high, overheads. On the contrary, a FNP system avoids preemption overhead at the cost of adding significant blocking delays due to lower priority tasks, affecting system schedulability.

To reduce the run-time overhead due to preemptions while still preserving the schedulability of the system, the limited preemption (LP) approach has been proposed in the literature [8][9][10]. In the LP model, tasks include *non-preemptive regions* (NPRs), where preemptions from other tasks are disabled. Depending on the location of the NPRs, fixed or floating, different schedulability analysis have been proposed. In this paper, we will focus on the *fixed NPR model*, where tasks preemption is allowed only at predefined locations inside the code, called *preemption points*, which divide tasks into fixed NPRs or sub-tasks. If higher priority tasks arrive between two preemption points of the running task, preemption is postponed until the next preemption point. The benefits of this model are multiple: (i) the number of preemptions is reduced; (ii) a tighter analysis of the preemption-related overhead is possible; and (iii) the preemption overhead may be significantly reduced by an optimized placement of preemption points [11].

Within the fixed LP model, there are two approaches which differ in the way in which the lower-priority running task to preempt is determined. The *Eager* LP approach selects to preempt the lower-priority running task that first reaches a preemption point. The *Lazy* LP approach waits until the lowest priority running task reaches a preemption point. So far, these two approaches have been analyzed only for sequential task-sets [12][13]. Under this setting, the eager approach causes high-priority tasks to potentially experience multiple priority inversions due to lower priority tasks. In the lazy approach, instead, a task may only be blocked at the beginning of its execution and so suffering one single priority inversion, although for a potentially larger amount of time than in the eager case. The two approaches are therefore incomparable, meaning that there are sequential task-sets that can be scheduled only with

This work was funded by the EU projects P-SOCRATES (FP7-ICT-2013-10) and HERCULES (H2020/ICT/2015/688860), and the Spanish Ministry of Science and Innovation under contract TIN2015-65316-P.

eager but not with lazy, and viceversa [13].

The analysis for sequential tasks cannot be applied as-it-is when considering DAG-based task-sets. A lazy scheduler, for example, may cause a DAG-task to experience multiple priority inversions due to its inherent parallel structure. As we will explain, the variation in the number of cores occupied by a DAG-task may allow lower priority tasks to repeatedly block later nodes of higher priority tasks, increasing the overall interference.

In this paper, we analyze these effects, provide a sufficient schedulability analysis for both the eager and the lazy approaches, and compare their performance in terms of number of priority inversions, preemptions and overall schedulability.

Concretely, this paper provides the following contributions: (1) It analyzes and formally proves the conditions under which a DAG-task may experience priority inversion with global fixed priority LP scheduling; (2) It provides a sufficient response time analysis for the *eager* approach, highlighting and correcting a subtle error found in a recent paper addressing the same setting [14]; (3) It derives a novel response time analysis for DAG-based task systems globally scheduled with a *lazy* approach, reducing the number of priority inversions that may be experienced with respect to an eager scheduler; (4) It shows that, despite the eager approach generates a higher number of priority inversions, the blocking impact of lower-priority tasks is significantly lower, particularly when the number of cores increases; (5) It shows how the lower-priority blocking impacts the overall schedulability performance, evaluating the proposed tests for eager and lazy approaches on both, a randomly generated parallel workloads and a real automotive AUTOSAR application, i.e., a diesel engine management system (EMS) provided by DENSO.

II. Related Work

In order to reconcile the massively parallel computation capabilities of modern multi- and many-core architectures with the timing requirements of embedded computing applications, parallel programming models are increasingly studied by the real-time community. Recently, new task models have been proposed to resemble the fine-grained execution provided by current parallel programming paradigms. Among the proposed task models, there are the *fork-join* model [15], in which each task is represented as a sequence of alternating sequential and parallel segments, and the *synchronous parallel* model [16][17], which allows consecutive parallel segments with an arbitrary degree of parallelism. The sporadic DAG model [18][19] generalizes the two previous models by representing a task as a directed acyclic graph, where each node corresponds to a sequential piece of code, and edges represent precedence constraints between pairs of nodes. Most recently, the DAG model has been extended [20][21] to encompass both conditional and parallel execution modes. However, despite the significant amount of work on parallel task models, most of the existing literature on the topic only considers fully preemptive or fully non-preemptive scheduling strategies.

The potential of limited preemptive scheduling schemes has been mostly investigated in the case of sequential task-sets, for which they have been proven to effectively limit the preemption-related overhead incurred by a task. The reader can refer to [10] for an exhaustive survey on the limited preemptive scheduling framework in a single-core scenario.

In a multi-core system, schedulability analysis have been developed under both the lazy and eager approaches. In the former class, an analysis based on link-based scheduling has been proposed in [22], while a schedulability analysis targeting global fixed priority scheduling with eager preemptions has been proposed in [23], under the assumption that each task has a single final non-preemptive region. This work also showed that an appropriate choice of the length of this region can improve schedulability. Moreover, the authors showed that the limited preemptive approach under global fixed priority scheduling with eager preemptions is incomparable to that with lazy preemptions. A complete schedulability analysis in the case of eager preemptions has been proposed in [12].

To the best of our knowledge, the only work that extends the limited preemptive scheduling scheme to a parallel task model is [14], which proposed a response time analysis for the sporadic DAG task model considering limited preemptions under the eager approach.

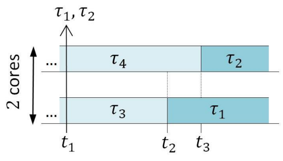
III. System Model

In this paper, we consider a task-set composed of n sporadic DAG tasks $\mathcal{T} = \{\tau_1, \cdots, \tau_n\}$ executing according to global fixed-priority scheduling on a platform composed of m identical cores. Each task τ_k releases an infinite sequence of jobs with a minimum inter-arrival time of T_k time-units and a constrained relative deadline $D_k \leq T_k$. We assume tasks are ordered according to their decreasing unique priority, i.e., τ_i has a higher priority than τ_j if $i < j$. We denote as $hp(k)$ and $lp(k)$ the subsets of tasks with higher and lower priorities than τ_k, respectively.

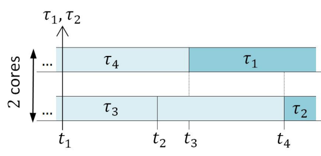
Each task $\tau_k \in \mathcal{T}$ is represented as a directed acyclic graph $G_k = (V_k, E_k)$. $V_k = \{v_{k,1}, \ldots, v_{k,q_{k+1}}\}$ is the set of nodes and $E_k \subseteq V_k \times V_k$ is the set of edges representing precedence constraints among pairs of nodes. If $(v_{k,1}, v_{k,2}) \in E_k$, then $v_{k,1}$ must complete before $v_{k,2}$ can begin execution. A node with no incoming arcs is referred to as a source of the DAG. Each node $v_{k,j} \in V_k$ is characterized by its *worst-case execution time* (WCET) $C_{k,j}$. Without loss of generality, we assume that each DAG task has exactly one source node $v_{k,1}$. If this is not the case, a dummy source node with zero WCET can be added to the DAG, with edges to all the source nodes.

Let $len(G_k)$ be the length of the longest path in the DAG, which also corresponds to the minimum amount of time needed to execute the DAG-task on a sufficiently large number of processors. Algorithms to compute this value in linear time are presented in the literature [20]. Moreover, let $vol(G_k)$ denote the volume of the DAG, i.e., the sum of the WCETs of all its constituting nodes. This value corresponds to the WCET of the task when executing on a dedicated single-core platform.

We consider a fixed priority scheduler with limited preemptions and fixed preemption points. We assume preemption points be given by node's boundaries, i.e., a task cannot be preempted while executing within a node, resembling the non-preemptive regions of an OpenMP program [24], i.e. "task parts" in the OpenMP nomenclature. Therefore, a task τ_k has $q_k = |V_k| - 1$ potential preemption points and the nodes in V_k represent non-preemptive regions (NPRs). Within the LP scheduling model, we consider two different approaches: (1) the *Eager* approach, where a high priority task preempts the first lower priority executing task that encounters a preemption



(a) LP eager approach



(b) LP lazy approach

Fig. 1. Scheduling of sequential tasks τ_1, τ_2, τ_3 and τ_4 on a 2-core processor.

point, i.e., that reaches the end of a node; and (2) the *Lazy* approach, where a high priority task waits until the lowest priority executing task reaches a preemption point.

Figure 1(a) shows an example of the eager approach applied to the task-set $\{\tau_1, \tau_2, \tau_3, \tau_4\}$ executing on two cores. For simplicity, the task-set is composed of sequential tasks. Assume that tasks τ_3 and τ_4 are already executing when τ_1 and τ_2 are released at time t_1. Under the eager preemption approach, τ_1 starts executing as soon as the lower priority task τ_3 reaches a preemption point, which occurs at time instant t_2. Similarly, task τ_2 starts the execution at time instant t_3 when the next lower priority task τ_4 reaches a preemption point.

With a lazy approach instead, τ_1 waits until τ_4, the lowest priority running task, reaches a preemption point at time t_3, as shown in Figure 1(b). Notice that another lower priority task (τ_3) reached a preemption point befire, at t_2, but has not been preempted. Hence, task τ_2 is blocked by τ_3 until this latter reaches its next preemption point at time t_4.

IV. Schedulability Analysis

To investigate the schedulability analysis of DAG-based task systems with limited preemptions, we build upon the work in [14] that extends the analysis for fully-preemptive systems presented in [20] by incorporating the interference due to non-preemptive regions of lower-priority tasks. A response-time analysis of a DAG-based task-set with a limited-preemptive global fixed priority scheduler is computed by iterating the following equation until a fixed point is reached, starting with $R_k = len(G_k) + \frac{1}{m}\big(vol(G_k) - len(G_k)\big)$:

$$R_k \leftarrow len(G_k) + \frac{1}{m}\big(vol(G_k) - len(G_k) + I_k^{hp} + I_k^{lp}\big) \quad (1)$$

To understand the above formula, we hereafter clarify the constituting terms: (1) $len(G_k)$ represents the length of the critical path of task τ_k, i.e., the sequence of consecutive nodes leading to the minimum response-time of τ_k. (2) $vol(G_k) - len(G_k)$ represents a valid upper-bound on the *self-interference* (or intra-task interference) of task τ_k, i.e., the interfering contribution experienced by the considered task due to nodes not belonging to its critical path. (3) I_k^{hp} represents the *higher-priority interference* from higher priority tasks in the system, i.e., the amount of time each higher priority task executes while τ_k is pending but not executing. Similarly, I_k^{lp} represents the *lower-priority interference*. The sum $I_k^{hp} + I_k^{lp}$ is also known as inter-task interference. (4) The intra-task and inter-task interfering contribution can be divided by m by observing that the critical path of τ_k does not make any progress *only when all cores are occupied by intra-task and/or inter-task interference* (see Lemma IV.2 in [20]).

To apply the above analysis, it is necessary to identify valid upper bounds on the terms I_k^{hp} and I_k^{lp}. The following

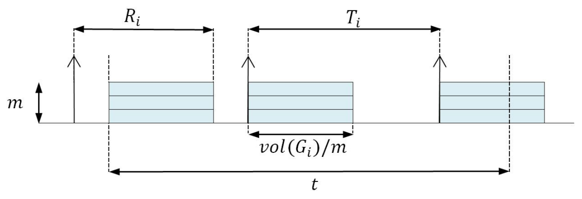


Fig. 2. Worst-case workload of a task τ_i in a window on length L.

Lemma, rephrased from [20], provides an upper bound for the higher-priority interference I_k^{hp}, by taking the densest possible packing of parallel jobs for each task in $hp(k)$, given a problem window of length R_k.

Lemma 1 (From [20]): An upper-bound on the higher-priority interference on task τ_k in a window of length R_k is given by

$$I_k^{hp} \leq \sum_{i \in hp(k)} \mathcal{W}_i(R_k) \quad (2)$$

where $\mathcal{W}_i(t) = \left\lceil \frac{t + R_i - vol(G_i)/m}{T_i} \right\rceil vol(G_i)$.

$\mathcal{W}_i(t)$, represented in Figure 2, is the maximum workload of an interfering task τ_i in a window of length t. It happens when (i) the volume of the higher priority task τ_i is evenly divided among all m cores; (ii) the carry-in job executes as late as possible, i.e., close to its worst-case response time which is upper-bounded by R_i; and (iii) later instances execute as soon as possible, i.e., when they are released with the minimum inter-arrival time. By considering a full contribution of both carry-in and carry-out instances, the lemma follows[1].

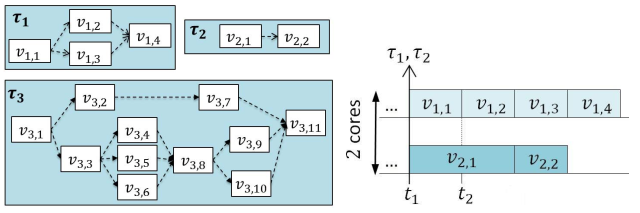
In the following sections, we will show how to provide valid upper bounds on the lower-priority interference I_k^{lp} for the Eager and the Lazy approaches.

V. Lower-Priority Interference

To compute an upper bound on the interference from lower priority tasks, it is first necessary to identify the situations in which a priority inversion may occur, i.e., when a task may be blocked by lower priority instances. As noted in previous works addressing global LP schedulability analysis [23], [12], [14], a task may be blocked before the beginning of its execution by lower priority tasks that already started executing and it may also suffer additional priority inversions due to later lower-priority instances.

To understand why a task may be blocked after it started executing, consider a sequential task system with eager preemptions. In this setting, a task τ_k may start executing along with one or more lower priority instances $\tau_{i>k}$. If a higher priority task arrives, τ_k may be preempted if it is the first one reaching a preemption point, even if it is not the lowest priority executing task. This would cause τ_k to be de-scheduled, while tasks $\tau_{i>k}$ continue executing, leading to a priority inversion until one of the lower priority executing tasks reaches a preemption point. Such a priority inversion is therefore bounded by the length of the NPRs of lower priority tasks.

[1]The corresponding Lemma V.1 in [20] does not consider a full carry-out contribution, but only the share that fits the considered problem window. However, we found that such a tighter estimation does not improve the analysis, since the response-time iteration will always continue until a full carry-out instance is considered. This observation allowed us to simplify the formula without introducing pessimism.



(a) Task-set composed of 3 DAG-tasks. (b) Scheduling of task τ_1 and τ_2.

Fig. 3. Example of DAG task-set.

However, for DAG-based tasks the situation is a bit more complicated. This is due to the inherent parallelism of the structure of each task, which may dynamically vary depending on the graph dependencies. Indeed, it may happen that an executing task τ_k may experience lower-priority blocking even without being preempted by higher priority tasks, e.g., when a node of τ_k forks two or more parallel nodes, requiring additional cores to execute them. If cores are busy executing lower priority instances, τ_k experiences blocking on the forked nodes until lower-priority instances reach a preemption point.

Figure 3 shows an example of this scenario. We consider a system composed of two DAGs, τ_1 and τ_2, executing on $m = 2$ cores. The structure of τ_1 and τ_2 is as in Figure 3(a), assuming all nodes have unitary WCET, except $v_{2,1}$ with $C_{2,1} = 2$. Task τ_1 is the highest priority task, therefore it cannot be preempted after it started executing. Still, τ_1 may be blocked by τ_2 once τ_1 has started its execution. This is the case, for instance, shown in Figure 3(b), when both tasks arrive at the same time t_1. When $v_{1,1}$ finishes, only one of the forked nodes $v_{1,2}$ may be scheduled at t_2, while $v_{1,3}$ will be blocked by the lower priority task τ_2 executing on the other core.

To identify the additional number of priority inversions once a task τ_k has started its execution and due to the dynamic variation of cores required, we introduce a new parameter sw_k.

Definition 1: sw_k is the maximum number of additional core requests that a DAG-task τ_k may cause after starting its execution.

These are all the points at which a potential priority inversion may arise after τ_k has started executing. It is important not to confuse the *extra* cores accounted by sw_k, with the number of spawns, the total spawned nodes, or the maximum cores required by a task simultaneously (maximum parallelism). To better understand the reasoning behind this definition, consider the example in Figure 3(a), where a DAG task-set composed of three tasks is represented. For the first task τ_1, $sw_1 = 1$ because one extra core is required when node $v_{1,1}$ spawns nodes $v_{1,2}$ and $v_{1,3}$. Notice that the maximum parallelism is 2 but the number of additional cores required is just 1. For task τ_2, $sw_2 = 0$ because it does not require extra cores after it started executing. Finally, $sw_3 = 4$ because τ_3 requires: (i) 1 extra core when node $v_{3,1}$ spawns nodes $v_{3,2}$ and $v_{3,3}$ ($sw_3 = 1$); (ii) 2 extra cores when node $v_{3,3}$ spawns nodes $v_{3,4}$, $v_{3,5}$ and $v_{3,6}$ ($sw_3 = 1 + 2$); and (iii) 1 extra core when node $v_{3,8}$ spawns nodes $v_{3,9}$ and $v_{3,10}$ ($sw_3 = 1+2+1 = 4$). Notice that, even though τ_3 occupies three cores when nodes $v_{3,4}$, $v_{3,5}$ and $v_{3,6}$ are ideally executed in parallel, two cores are released before executing $v_{3,8}$. Therefore, we need to account for an extra core when $v_{3,8}$ spawns nodes $v_{3,9}$ and $v_{3,10}$.

A linear algorithm that computes an upper bound on sw_k could be easily obtained by iterating over all nodes of each DAG, $v_{k,j} \in V_k$, and adding the number of $v_{k,j}$'s successors minus 1. However, this algorithm would be too pessimistic as there may be dependencies among successor nodes that prevent them to simultaneously execute, thus, reducing the number of extra cores required. For example, if an edge would exist from $v_{3,4}$ to $v_{3,5}$ in Figure 3(a), the number of extra cores required after the computation of $v_{3,3}$ is 1 and not 2 ($v_{3,3}$'successors minus 1) because $v_{3,4}$ should execute before $v_{3,5}$.

Algorithm 1 Additional core requests caused by a DAG task.

Input: $G = (V, E)$; $Succ(v_i)\ \forall\ v_i \in V$; $Pred(v_i)\ \forall\ v_i \in V$
Output: sw

```
 1: procedure AdditionalCores
 2:     $sw = 0$
 3:     $N = \{\}$
 4:     for each $v_i \in V$ do
 5:         $cores = |Succ(v_i)| - 1$
 6:         for each $v_j \in Succ(v_i)$ do
 7:             if $v_j \in N$ then
 8:                 $cores = cores - 1$
 9:             else
10:                 if $Succ(v_i) \cap Pred(v_j) \neq \phi$ then
11:                     $cores = cores - 1$
12:                 end if
13:                 $N = N \cup \{v_j\}$
14:             end if
15:         end for
16:         $sw = sw + max(0, cores)$
17:     end for
18: end procedure
```

Therefore, in order to compute the exact value of sw_k we provide an algorithm that considers potential dependencies among successor nodes. Concretely, Algorithm 1 takes as input the DAG $G = (V, E)$ and, for each node $v_i \in V$, $Succ(v_i)$ and $Pred(v_i)$ which are the sets of successors and predecessors of v_i, respectively. The algorithm iterates over all the nodes in V. At each iteration the number of extra cores required after v_i execution, i.e. the variable $cores$, is initialized to its maximum possible value: v_i's successors minus 1 (line 5). In the second loop, the algorithm iterates over all v_i's successors, $v_j \in Succ(v_i)$, to check if a core has already been accounted for the execution of v_j (line 7) or if it has a dependency with any of its sibling nodes (line 10). In both cases the number of extra cores required decreases by one. Then, the given successor v_j is added to N (line 13) to keep track of the nodes that have been already considered for accounting extra cores. Finally, sw is updated with the extra cores required after v_i execution (line 16). This algorithm has quadratic complexity in the number of nodes.

The computation of sw_k allow us to provide, in the following sections, the number of priority inversions, and therefore an upper bound on the lower-priority interference, to any DAG-task scheduled with the eager or lazy LP approaches.

VI. Eager preemption analysis

With the eager approach, the first lower priority task to reach a preemption point may be preempted even if it is not the lowest priority running task. That is, if there are several lower priority tasks running when a high priority task is released, the first lower priority task τ_k reaching a preemption point is preempted. As a result, τ_k can suffer lower priority interference not only before starting its execution, but also at later preemption points. Moreover, with a DAG-task model, τ_k may also suffer lower priority interference when it requires one or more additional cores for executing its parallel nodes.

The next lemma provides an upper bound on the number of higher-priority instances that may arrive within the scheduling window of a job of a task τ_k.

Lemma 2: In any time interval of length t, a job of task τ_k may be preempted by higher-priority tasks at most h_k times:

$$h_k(t) = \sum_{i \in hp(k)} \left\lceil \frac{t + R_i}{T_i} \right\rceil (1 + sw_i) \qquad (3)$$

Proof: Assume the job of τ_k is released at time $t_0 = 0$. During a time interval of length t, each higher-priority task τ_i can be released at most $\left\lceil \frac{t+R_i}{T_i} \right\rceil$ times. Following the definition of sw_i in the previous section, it descends that the number of cores requests (and so potential preemption to τ_k) by a single instance of τ_i is at most $1 + sw_i$: one when τ_i releases plus sw_i after it starts executing. If we consider all the high priority tasks in $hp(k)$, the lemma simply follows. ■

To determine the number of priority inversions experienced with the eager approach, the following lemma identifies the conditions under which an executing task experience lower-priority interference.

Lemma 3: Under the LP eager approach, a DAG-task τ_k that already started executing may experience additional lower-priority interference only if all the following conditions are simultaneously satisfied:

1) τ_k encounters a preemption point.
2) A higher priority task arrives OR τ_k requires extra cores to spawn new parallel nodes.
3) There are lower-priority tasks being executed.

Proof: Condition (1) guarantees that, following the LP scheduling model, a task cannot be preempted within the execution of a node. Conditions (2) follow from the observation that a task cannot be preempted by a lower-priority instance. Therefore, in order for τ_k to experience blocking from other lower-priority running instances, it must either be preempted by a higher-priority task *or* require extra cores to spawn new parallel nodes. Condition (3) is trivially derived by noticing that no lower-priority blocking may be experienced without lower-priority instances being executed. ■

Lemma 3 enables upper-bounding the maximum number of priority inversions for the eager approach as follows.

Lemma 4: With the LP eager approach, in any time interval of length t, an upper bound on the number of additional priority inversions that a DAG-task may experience after starting its execution is

$$p_k^{eager}(t) = min\Big(q_k, sw_k + h_k(t), \sum_{\forall \tau_i \in lp(k)} \left\lceil \frac{t + R_i}{T_i} \right\rceil \times |V_i|\Big) \qquad (4)$$

Proof: Condition (1) in Lemma 3 ensures that the number of additional priority inversions cannot exceed the number of potential preemption points predefined by the internal structure of the DAG (q_k). Condition (2) provides a further bound given by the number of preemption requests from higher priority instances during a time interval of length t ($h_k(t)$, see Equation 3) plus the overall number of extra cores requests by τ_k in spawn operations (sw_k). Finally, Condition (3) allows deriving one last bound given by the number of NPRs (i.e., nodes) of the lower priority tasks that may arrive within the considered scheduling window of length t $\left(\sum_{\forall \tau_i \in lp(k)} \left\lceil \frac{t+R_i}{T_i} \right\rceil \times |V_i|\right)$. ■

The *lower priority interference* I_k^{lp} for the LP eager approach has been computed in [14] as:

$$I_k^{eager} = \Delta_k^m + p_k^{eager}(R_k) \times \Delta_k^{m-1} \qquad (5)$$

where (considering the *LP-max* approach [14]):

$$\Delta_k^m = \sum_{l=1}^{m} Q_k^l \qquad \text{and} \qquad \Delta_k^{m-1} = \sum_{l=1}^{m-1} Q_k^l \qquad (6)$$

Δ_k^m upper-bounds the lower priority interference on the first NPR of τ_k, while Δ_k^{m-1} upper-bounds the lower priority interference on the p^{th} NPRs ($2 \leq p \leq q_k + 1$) of τ_k, i.e. when a priority inversion occurs. Q_k^l denotes the l^{th} *largest NPR* in the set $lp(k)$. In order to upper bound the lower-priority blocking suffered by a task τ_k, the analysis in [14] considers that τ_k's lower priority tasks are executing their m longest NPRs on the m available cores when τ_k is released. Moreover, in the worse case, at each τ_k's potential priority inversion (due to a higher priority tasks or because τ_k requires extra cores), the $m-1$ longest NPRs of τ_k's lower priority tasks are executing on the $m - 1$ available cores. In [12], authors demonstrate why only $m - 1$ lower priority tasks may block τ_k at each preemption.

Note that we enhanced the derivation of p_k^{eager} with respect to [14]. On one side, Equation 4 includes the potential blocking impact of spawn operations (sw_k). Not considering it underestimates the number of priority inversions, as the potential blocking impact when extra cores are requested is not taken into account. On the other side, p_k^{eager} includes the maximum number of nodes coming from lower priority tasks that may interference with τ_k ($\sum_{\forall \tau_i \in lp(k)} \left\lceil \frac{t+R_i}{T_i} \right\rceil \times |V_i|$), resulting in a more accurate blocking estimation.

VII. Lazy preemption analysis

Under the lazy approach, preemption is delayed until the lowest priority currently running task reaches a preemption point. Authors in [25] estimated the worst-case interference due to lower-priority tasks when considering sequential task-sets and lazy preemptions. Under this scenario, a high priority task τ_k may only suffer from lower priority interference before it starts its execution, i.e., when it is released. With sequential task-sets and the lazy approach, τ_k can only be preempted by higher priority tasks if it is the lowest priority task executing on the processor. As a result, τ_k cannot be interfered by lower priority tasks once it has started executing.

Figure 4 illustrates an example of the worst-case blocking pattern generated by lower priority tasks under a sequential task-set scenario. Concretely, it shows 8 tasks $\tau_1, ..., \tau_8$ (in decreasing priority order) running on $m = 4$ cores. Assume that lower-priority tasks τ_5, τ_6, τ_7 and τ_8 are already executing

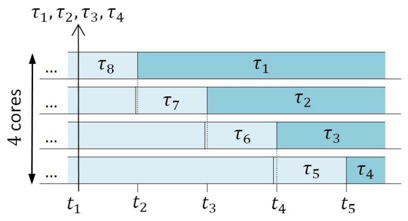


Fig. 4. Maximum lower priority blocking of a sequential task-set under the lazy approach.

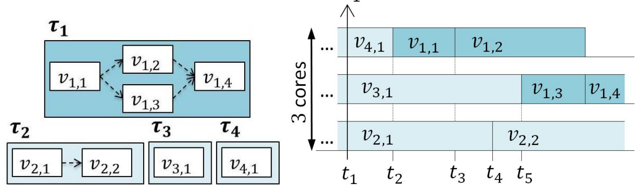
on the processor, when the higher-priority tasks τ_1, τ_2, τ_3 and τ_4 are simultaneously released at time instant t_1. The first task to be preempted is τ_8 (the lowest priority task) at time t_2, when a preemption point is reached, and so the highest priority ready task τ_1 can start its execution. In the worst case scenario task τ_7 reaches a preemption point at time $t_2 - \varepsilon$ in which the lowest priority task (τ_8) is still executing, and so it continues executing (an so blocking τ_2) until its next preemption point is reached at t_3, when τ_2 can start executing. Subsequently in the worst-case situation, τ_6 enters in a node just before the preemption point of τ_7 is reached at time t_3, blocking τ_3 until t_4. Finally, τ_4 is able to start its execution at time t_5. Overall, the worst-case blocking τ_4 can suffer is equal to $(t_2{-}t_1){\times}4{+}(t_3{-}t_2){\times}3{+}(t_4{-}t_3){\times}2{+}(t_5{-}t_4){\times}1$. In general, the upper bound of the maximum blocking is computed by adding the largest node from the $lp(k)$ multiplied by m, the second largest node from the $lp(k)$ multiplied by $m-1$, the third largest node from the $lp(k)$ multiplied by $m-2$, and so on, until the m-th largest nodes from $lp(k)$ is considered.

However, with a DAG-based task system, the lazy approach involves not only suffering low priority blocking on the first node, i.e., when the task is released, but also at other intermediate execution points, namely, *when extra cores are required to spawn new parallel nodes.*

Figure 5 shows an example of the lazy approach when considering a DAG-based task-set composed of four tasks τ_1, τ_2, τ_3 and τ_4, in decreasing priority order (Figure 5(a)), scheduled on $m = 3$ cores (Figure 5(b)). We assume that tasks τ_2, τ_3 and τ_4 are executing its first node when the highest-priority task τ_1 is released at time t_1. Under the lazy approach, τ_1 starts executing the first node $v_{1,1}$ when the lowest-priority running task τ_4 reaches a preemption point at time t_2. At time t_3, nodes $v_{1,2}$ and $v_{1,3}$ are ready to start executing, but only the core in which $v_{1,1}$ is being executed is available to start executing $v_{1,2}$. At time t_4, τ_2 reaches a preemption point, but since it is not the lowest priority task, node $v_{1,3}$ of τ_1 is blocked until τ_3 reaches a preemption point at time t_5. As a result, when considering DAG-based task-sets, *intermediate nodes can suffer from low priority blocking*. In the example, the lower priority task τ_2 and τ_3 block the execution of the intermediate node $v_{1,3}$. The reason is because, at time t_3, τ_1 spawned two parallel nodes, and so requested one extra core.

Lemma 5: Under the LP lazy approach, a DAG-task τ_k that already started executing may experience additional lower priority interference only if all the following conditions are simultaneously satisfied:

1) τ_k encounters a preemption point
2) τ_k requires extra cores to spawn new parallel nodes
3) There are lower-priority task being executed



(a) Task-set composed of 4 DAGs. (b) Scheduling on a 3-core processor.

Fig. 5. Scheduling of a DAG-based task-set under the lazy approach.

Proof: Similarly to Lemma 3, Condition (1) guarantees that, following the LP scheduling model, a task cannot be preempted within the execution of a node. Condition (2) follows from the observation that τ_k can only be preempted by a higher-priority task and if it is the case, following the lazy approach, τ_k would be the lowest priority tasks (there are not other lower priority instances running and blocking τ_k). However, if τ_k requires extra cores to spawn new parallel nodes, all cores may be occupied by lower priority tasks blocking τ_k. Condition (3) is trivially derived by noticing that no lower-priority blocking may be experienced without lower-priority instances being executed. ■

Lemma 5 enables upper-bounding the maximum number of priority inversions for the lazy approach as follows:

Lemma 6: With the LP lazy approach, in any time interval of length t, an upper bound on the number of additional priority inversions that a DAG-task may experience after starting its execution is

$$p_k^{lazy}(t) = min\Big(sw_k, \sum_{\forall \tau_i \in lp(k)} \left\lceil \frac{t + R_i}{T_i} \right\rceil \times |V_i|\Big) \quad (7)$$

Proof: Similarly to the eager approach, the number of additional priority inversions is upper bounded by the number of extra cores requests sw_k (see Definition 1) and the number of lower priority tasks' nodes $\left(\sum_{\forall \tau_i \in lp(k)} \left\lceil \frac{t+R_i}{T_i} \right\rceil \times |V_i|\right)$. Despite Lemma 5 defines as a necessary condition to encounter a preemption point, $q_k > sw_k$ so, additional priority inversions are not conditioned to a preemption point but to a spawn operation. To demonstrate it, consider a DAG-task with a node spawning m different nodes. In this case, $q_k = m$ while $sw_k = m - 1$ as the same core executing the first node can execute one of the spawned nodes. ■

Overall, and following the approach in [14], the *lower priority interference* I_k^{lp}, for the LP lazy approach is computed as:

$$I_k^{lazy} = A_k^m + p_k^{lazy}(R_k) \times A_k^{m-1} \quad (8)$$

where (considering the blocking estimation presented in [25], named ADS blocking estimation 2):

$$A_k^m = \sum_{l=1}^{m} Q_k^l \times (m - l + 1) \text{ and } A_k^{m-1} = \sum_{l=1}^{m-1} Q_k^l \times (m - l) \quad (9)$$

As shown in the previous Section VI, Q_k^l denotes the l^{th} *largest NPR* in the set $lp(k)$.

VIII. Evaluation

This section evaluates the proposed response time analysis of both the eager and lazy limited preemption approaches

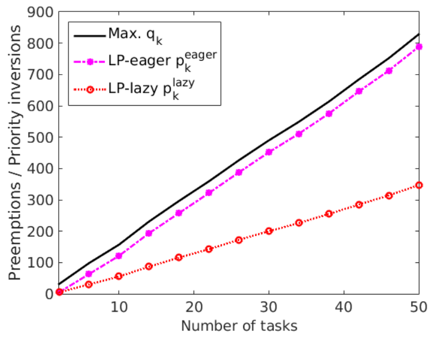


Fig. 6. Number of additional priority inversions of LP-eager (p_k^{eager}) and LP-lazy (p_k^{lazy}) and DAG's maximum number of preemptions (q_k), when varying the number of tasks.

(labeled as *LP-eager* and *LP-lazy* respectively) in terms of: (1) number of priority inversions considered by our response time analysis according to Equations 4 and 7, (2) number of preemptions occurring at system deployment (by means of a scheduling simulator), (3) impact of interference and blocking coming from high-priority and low-priority tasks respectively, and (4) schedulability analysis varying the overall system utilization and number of tasks.

It is important to remark that the schedulability test is an iterative procedure which computes the response time upper bound of each task, starting from the highest priority task to the lowest priority tasks. Therefore $R_i, \forall \tau_i \in lp(k)$, is not computed yet when, iterating over task k, we need to compute p_k^{eager} or p_k^{lazy} (see equations 4 and 7). As a result, in this section we use a safe upper bound of R_i, which is D_i (if D_i is greater than R_i then the task set is not schedulable).

LP-eager and LP-lazy are compared against an ideal FP (labeled as *FP-ideal*) in which the impact of lower-priority blocking is discarded in Equation 1, i.e. I_k^{lp} equals to 0. Notice that the performance of a real FP strategy in which the preemption overheads would be included in the analysis may significantly decrease compared to LP. Accurately accounting for preemption overheads in FP is very difficult (if not impossible) since the execution can be preempted at any point of the task. Preemption overheads have not been considered in LP-eager nor in LP-lazy because they have the same impact in the response time analysis of both strategies. Nevertheless, a safe upper bound could be easily computed by multiplying the maximum number of preemptions a task may suffer, q_k, by the maximum time required for a context switch.

All results have been implemented in MATLAB®, considering the simulation environment presented in [14].

A. Case-studies: Randomly Generated Task-sets and an AUTOSAR Automotive Application

Experiments presented in Sections VIII-B and VIII-C consider random DAG-based task-sets generated with the simulation environment presented in [14], assuming the following parameters: the maximum number of nodes (NPRs) per graph is $V_{max} = 50$; the probability of creating a terminal node or keeping the expansion of the graph are $p_{term} = 0.4$ and $p_{par} = 0.6$, respectively; the probability of adding an additional edge between sibling nodes is $p_{dep} = 0.1$; the maximum number of successors a node can have is $n_{par} = 6$; the longest path of the DAGs is at most 7; the WCET $C_{i,j}$ of each node is uniformly selected in the interval $[1, 100]$. For each experiment, we generated 500 task-sets for each value in the x-axis, considering the implicit deadline case ($D_k = T_k$).

Experiments presented in Section VIII-D consider an AUTOSAR-compliant diesel Engine Management System (EMS). AUTOSAR is a standardized system software architecture upon which automotive applications are built and executed [26]. In AUTOSAR, applications are composed of a set of functions, named runnables, that communicate among them through well-defined communication methods. Runnables, that can be executed periodically or triggered by an interrupt, are grouped into AUTOSAR tasks, which are the unit of scheduling (UoS) of the AUTOSAR Operating System. The nature of AUTOSAR execution model fits very well the preemptive scheduling model [27] considered in this paper: an AUTOSAR task can be modeled as a DAG-task where nodes correspond to runnables and edges correspond to communication methods among runnables. Runnables are executed uninterruptedly, defining preemption points at runnable boundaries.

An EMS is a typical automotive complex application in which the amount of fuel and the injection time are fundamental for smooth revolutions of the engine. To do so, the EMS requires an update either from eleven cyclic executed AUTOSAR tasks, with periods (and deadlines) of 1, 4, 5, 8, 16, 20, 32, 64, 96, 128 and 1024 ms, and one crank-angle triggered task, with a minimum period of 1.25 ms. The crank-angle task has a period that varies depending on the revolutions of the engine, so with the objective of guaranteeing the correct execution of the task, the smallest possible period is considered for scheduling purposes. Overall, the EMS is composed by more than one thousand runnables.

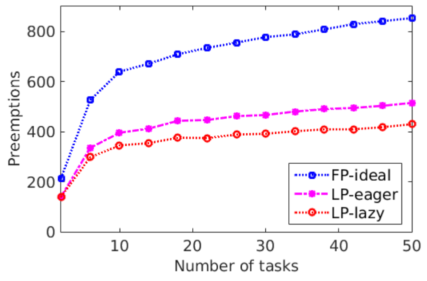
In order to compute the WCET estimates (C) of runnables, we use a static timing analysis tool OTAWA [28], which models a generic multi-core processor architecture. Concretely, we consider 4-core, 8-core and 16-core processor setup with private per-core scratchpads for instructions and write-through data caches. For all processor configurations cores are connected through a tree NoC to the on-chip RAM memory. The impact of interferences resultant of accessing to shared processor resources are not considered in the WCET computation[2].

B. Impact of Priority Inversions and Preemptions

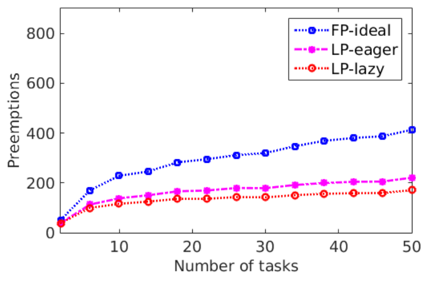
This section evaluates the number of preemption points and priority inversions considered at response time analysis, the number of actual preemptions occurring at system deployment, and the interference and blocking impact generated by high-priority and low-priority tasks respectively. All experiments presented in this Section consider randomly generated task-sets with an overall task-set utilization of 2.5.

Figure 6 shows the number of additional priority inversions considered by the LP-eager and LP-lazy strategies, i.e. p_k^{eager} (Equation 4) and p_k^{lazy} (Equation 7), and q_k the number of preemption points, i.e. the maximum number of preemptions a task may suffer, when varying the number of tasks from 2 to 50 (in steps of 4). Notice that q_k represents the maximum number of priority inversions after starting its execution.

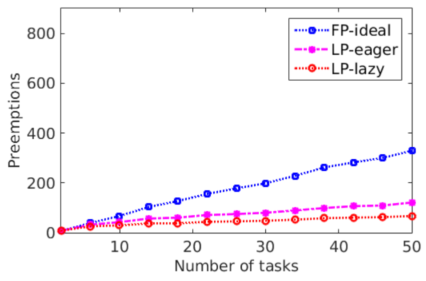
[2]The approach presented in this paper is independent of the processor architecture and the timing analysis method, so other architectures and tools can be used to compute the WCET estimates of runnables.



(a) $m = 4$ cores

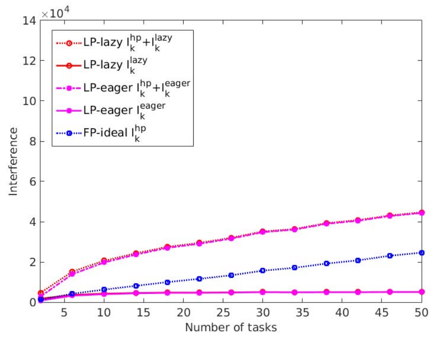


(b) $m = 8$ cores

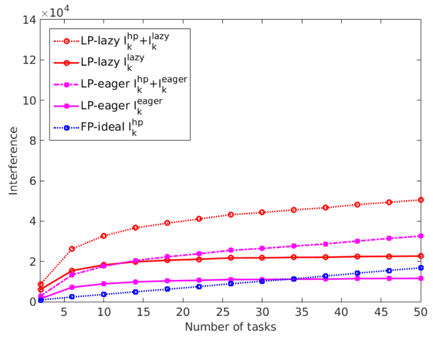


(c) $m = 16$ cores

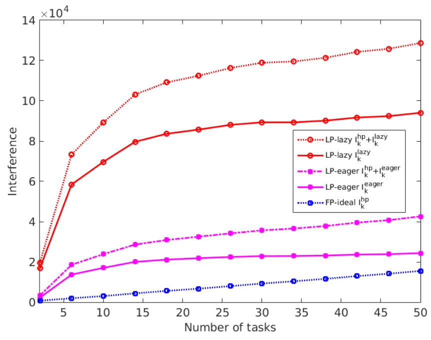
Fig. 7. Observed preemptions when varying the number of tasks and considering 4 (a), 8 (b) and 16 (c) cores.



(a) $m = 4$ cores



(b) $m = 8$ cores



(c) $m = 16$ cores

Fig. 8. Average higher- and lower-priority interference for each task when varying the number of tasks and considering 4 (a), 8 (b) and 16 (c) cores.

As expected, Figure 6 confirms that the response time analysis of the eager approach considers a higher number of potential priority inversions than the lazy approach, being very close to the maximum number of preemptions. The reason is that in the eager approach, lower priority blocking can come from (1) high-priority preemptions at the end of each node while there are lower priority task running and (2) the *spawn* of new parallel nodes requesting extra cores (see Lemma 3). Under the lazy approach instead, only *spawn* operations can generate blocking from lower priority tasks (see Lemma 5). In fact, in most cases, p^k_{eager} is given by q_k except for (1) the highest priority task, for which $p^k_{eager} = sw_k$ because $h_k = 0$ and $sw_k < q_k$, and (2) the lowest priority task, for which $p^k_{eager} = 0$ since there are no lower priority tasks causing blocking. The impact of these two tasks is shown in Figure 6, in the small difference between q_k and p^k_{eager}. In most cases, p^k_{lazy} is given by sw_k, except for the lowest priority task, for which $p^k_{lazy} = 0$ for the same reason than in LP-eager.

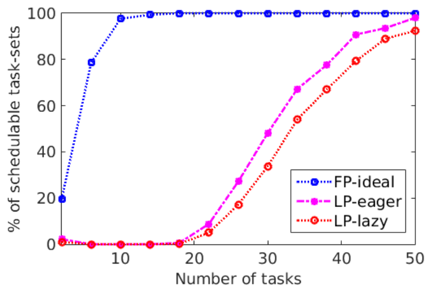
Such a trend is also observed when actually executing the task-sets. Figure 7 shows the observed preemptions when executing the DAG task-sets in a scheduling simulation running for 10^5 time units (which includes multiple task releases), varying the number of tasks from 2 to 50 (in steps of 4), and considering a 4-core (a), 8-core (b) and 16-core (c) processor. In this case, the FP scheduling strategy has been considered as well, for comparison purposes.

As expected, the LP eager approach generates more preemptions than the LP lazy approach. Clearly, the number of preemptions in both cases decreases as more cores are available for the same number of tasks. In case of the FP scheduling strategy, the number of preemptions is much higher than LP, since more scheduling opportunities exist (resulting in a higher schedulability rate, as will be shown in Section VIII-C). However, this would (seriously) difficult response time analysis if preemption overheads would be included.

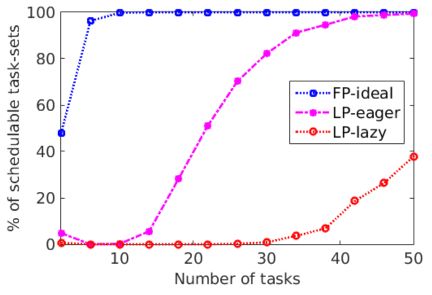
Despite LP-eager enforces a higher number of priority inversions compared to LP-lazy, as shown in Figure 6, it results in less blocking and so better schedulability ratio in the response time analysis (see Section VIII-C). Figure 8 shows the contribution (in time units) that interference and blocking due to high and low priority tasks make over the overall response time when varying the number of tasks from 2 to 50 (in steps of 4), and considering a 4-core (a), 8-core (b) and 16-core (c) processor. Concretely, the Figure shows the contribution of I^{eager}_k and I^{lazy}_k, and the sum of $I^{hp}_k + I^{eager}_k$ and $I^{hp}_k + I^{lazy}_k$, to the response time analysis of LP-eager and LP-lazy. The contribution of I^{hp}_k alone is also shown for FP-ideal.

As shown, the blocking factor due to lower priority tasks of LP-lazy (I^{lazy}_k) and LP-eager (I^{eager}_k) are almost equivalent when $m = 4$ cores but it increases dramatically as the number of cores increases. In case of $m = 16$ cores, I^{lazy}_k becomes the dominant factor in the response time. It is important to remark that the higher-priority interference I^{hp}_k is alike computed for FP, LP-eager and LP-lazy (see Equation 2). However, the I^{hp}_k for lazy is always worse than eager, which in turn, is worse than FP. The reason for this is because I^{hp}_k is computed considering the *window of interest* in which higher priority tasks can interfere, a.k.a the response time, which is iteratively computed by Equation 1. As I^{lazy}_k or I^{eager}_k increase, the window of interest increases as well, impacting on I^{hp}_k.

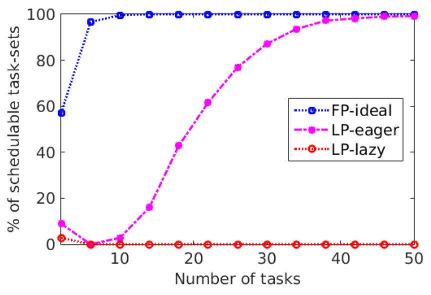
Overall, factors A^m_k and A^{m-1}_k (used in I^{lazy}_k) add huge pessimism, increasing the window of interest and impacting on the system schedulability of LP-lazy as shown in next section.



(a) $m = 4$ cores

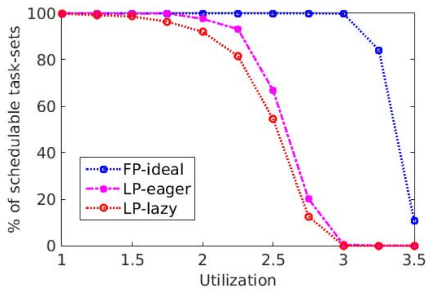


(b) $m = 8$ cores

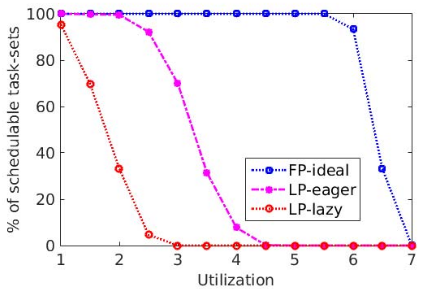


(c) $m = 16$ cores

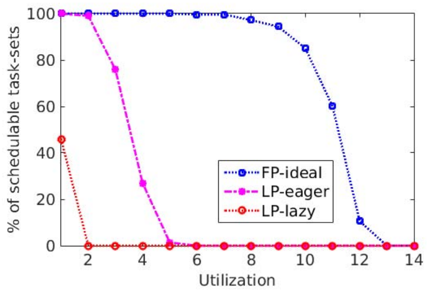
Fig. 9. Schedulability rate (in %) when varying the number of tasks and considering $m = 4$ (a), $m = 8$ (b) and $m = 16$ (c) cores.



(a) $m = 4$ cores



(b) $m = 8$ cores



(c) $m = 16$ cores

Fig. 10. Schedulability rate (in %) when varying the overall system utilization and considering $m = 4$ (a), $m = 8$ (b) and $m = 16$ (c) cores.

C. Schedulability Analysis

This section evaluates the schedulability rate (in percentage) resultant of the response time analysis presented in Section IV. The lower priority interference I_k^{lp} has been computed using Equations 5 (I_k^{eager}) and 8 (I_k^{lazy}) for the LP eager and lazy approaches respectively. All experiments presented in this Section consider randomly generated task-sets.

Figure 9 shows the percentage of schedulable task-sets for FP, LP-eager and LP-lazy when varying the number of tasks from 2 to 50 (in steps of 4), and considering a 4-core (a), 8-core (b) and 16-core (c) processor. The overall task-sets utilization is set to 2.5 for all cases. Since the number of tasks is a fixed parameter in each experiment, we computed individual utilization using UUnifast [29]. The common trend is that the schedulability ratio increases as the number of task increase conforming to the intuition that scheduling a large number of light tasks (with low individual utilization) is easier than scheduling fewer heavy tasks (with high individual utilization). This is the case of the FP-ideal and LP-eager for all processor configurations, with a schedulability rate of 100% (or very close to it) on task-sets composed of 50 DAG-tasks

However, for LP-lazy, I_k^{lazy} hugely increases as the number of cores increases (see Figure 8), resulting in a very pessimistic response-time analysis, in which no task-set can be scheduled, even with an utilization of 2.5 in a 16-core processor.

Figure 10 shows the percentage of schedulable task-sets for FP, LP-eager and LP-lazy when varying the task-set utilization and considering a 4-core (a), 8-core (b) and 16-core (c) processor. The task-sets considered for this experiments contains between 30 and 50 tasks. Results confirm what presented in previous figures: the impact of blocking due to low-priority tasks dominates on the response time analysis of the LP-lazy, resulting in a very poor schedulability rate. LP-lazy cannot schedule any task-set with values for the utilization above 3, 3 and 2, when considering 4, 8 and 16-cores respectively. For LP-eager, these utilization values are 3, 4.5 and 6.

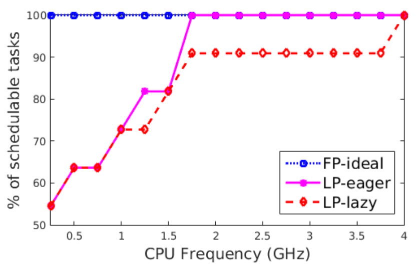
D. AUTOSAR Automotive case study

This section evaluates the schedulability of the EMS AUTOSAR application. We show the result of the response time analysis presented in Section IV for each of the twelve tasks.

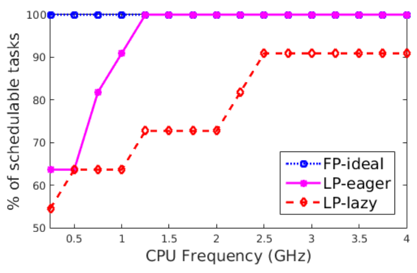
Figure 11 shows the percentage of schedulable tasks when ranging the CPU frequency from 250 MHz to 4 GHz and considering a 4-core (a), 8-core (b) and 16-core (c) processor. The reason of ranging among different processor frequencies is to evaluate the EMS application under different utilization scenarios. WCET estimations of runnables, i.e., nodes in V, are computed in CPU cycles, but tasks' periods are expressed in milliseconds (ms). Hence, the processor frequency must be considered to derive WCET estimations in the same time unit (ms). Moreover, increasing the CPU frequency is equivalent to decrease the overall task-set utilization. For example, when the processor operates at 250 MHz, the overall EMS utilization equals to 0.57; 4 GHz corresponds to an utilization of 0.03.

The trend shown in Figure 11 is similar to the one observed in Section VIII-C, i.e., LP-eager outperforms LP-lazy in all cases. In fact, LP-lazy cannot schedule the EMS application in any processor frequency configuration, except assuming a 4-core processor operating at 4 GHz. The pessimism added by LP-lazy increases as the number of cores increases, resulting in the counter-intuitive result where the schedulability decreases as the number of cores increases. Instead, under the LP eager approach, the EMS application is schedulable when the CPU frequency is equal or higher than 1.75 GHz, 1.25 GHz and 750 MHz for a 4, 8 and 16-core configuration respectively (with an overall utilization of $0.08, 0.11$ and 0.2). As expected, the schedulability increases as the number of cores increases.

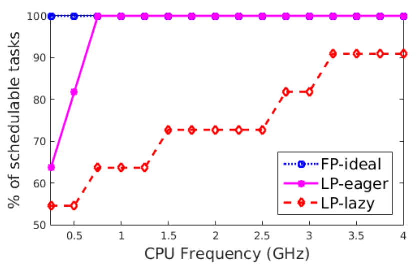
Overall, we conclude that LP-eager clearly outperforms



(a) $m = 4$ cores



(b) $m = 8$ cores



(c) $m = 16$ cores

Fig. 11. Percentage of schedulable tasks from the EMS AUTOSAR application when varying the CPU frequency and considering $m = 4$ (a), $m = 8$ (b) and $m = 16$ (c) cores.

LP-lazy, despite a higher number of priority inversions are considered in the response time analysis (see Figure 6), and a high number of preemptions are observed at system deployment (see Figure 7). In all cases, FP-ideal outperforms the LP-eager, as the blocking impact of low-priority tasks is not considered.

IX. Conclusions

With the advent of multi- and many-core embedded processors, DAG-based scheduling models are gaining a lot of attention due to its capability to effectively model the parallel execution. This paper evaluates the limited preemption (LP) strategy under global fixed priority for DAG-based task-sets. It shows the necessary conditions under which DAG tasks may experience lower priority blocking for the two identified preemption approaches: eager and lazy. Concretely, we formally proved which are these conditions and compute the number of priority inversions which leads to lower priority blocking. Finally, we evaluate and compare the response time analysis for the eager and lazy approaches with both, randomly generated task-sets and a AUTOSAR-compliant automotive application, i.e., a diesel engine management system (EMS). Our analysis demonstrates that, despite the eager approach generates a higher number of priority inversions, the blocking factor of the lazy approach dominates the response time upper bound. Therefore, contrary to what has been demonstrated when considering sequential task-sets, the LP lazy scheduling approach has been proven to be a very inefficient scheduling strategy when DAG-based task-sets are considered, and so not suitable for parallel execution.

References

[1] B. D. de Dinechin, D. van Amstel, M. Poulhies, and G. Lager, “Time-critical computing on a single-chip massively parallel processor,” in *DATE*, 2014.

[2] E. Stotzer, A. Jayraj, M. Ali, A. Friedmann, G. Mitra, et al., “OpenMP on the Low-Power TI Keystone II ARM/DSP SoC,” in *IWOMP*, 2013.

[3] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, “A generalized parallel task model for recurrent real-time processes,” in *RTSS*, 2012.

[4] “OpenMP API, Version 4.5,” OpenMP ARB, Tech. Rep., Nov. 2015.

[5] S. Baruah, M. Bertogna, and G. Buttazzo, *Multiprocessor Scheduling for Real-Time Systems*. Springer, 2015.

[6] B. B. Brandenburg, “Scheduling and locking in multiprocessor real-time operating systems,” Ph.D. dissertation, University of North Carolina at Chapel Hill, 2011.

[7] A. Bastoni, B. Brandenburg, and J. Anderson, “Cache-related preemption and migration delays: Empirical approximation and impact on schedulability,” in *OSPERT*, July 2010.

[8] R. J. Bril, J. J. Lukkien, and W. F. Verhaegh, “Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited,” in *ECRTS*, 2007.

[9] M. Bertogna and S. Baruah, “Limited preemption EDF scheduling of sporadic task systems,” *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 579–591, 2010.

[10] G. Buttazzo, M. Bertogna, and G. Yao, “Limited preemptive scheduling for real-time systems: A survey,” *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, March 2012.

[11] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo, “Optimal selection of preemption points to minimize preemption overhead,” in *ECRTS*, July 2011.

[12] A. Thekkilakattil, R. I. Davis, R. Dobrin, S. Punnekkat, and M. Bertogna, “Multiprocessor fixed priority scheduling with limited preemptions,” in *RTNS*, November 2015.

[13] A. Thekkilakattil, K. Zhu, Y. Nie, R. Dobrin, and S. Punnekkat, “An empirical investigation of eager and lazy preemption approaches in global limited preemptive scheduling,” in *Ada-Europe International Conference on Reliable Software Technologies*, 2016.

[14] M. A. Serrano, A. Melani, M. Bertogna, and E. Quinones, “Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions,” in *DATE*, March 2016.

[15] K. Lakshmanan, S. Kato, and R. Rajkumar, “Scheduling parallel real-time tasks on multi-core processors,” in *RTSS*, 2010.

[16] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. D. Gill, “Multi-core real-time scheduling for generalized parallel task models,” *Real-Time Systems*, vol. 49, no. 4, pp. 404–435, 2013.

[17] C. Maia, M. Bertogna, L. Nogueira, and L. M. Pinho, “Response-time analysis of synchronous parallel tasks in multiprocessor systems,” in *RTNS*, October 2014.

[18] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, “Feasibility analysis in the sporadic DAG model,” in *ECRTS*, July 2013.

[19] S. Baruah, “Improved multiprocessor global schedulability analysis of sporadic DAG task systems,” in *ECRTS*, July 2014.

[20] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo, “Response-time analysis of conditional DAG tasks in multiprocessor systems,” in *ECRTS*, July 2015.

[21] S. Baruah, V. Bonifaci, and A. Marchetti-Spaccamela, “The global EDF scheduling of systems of conditional sporadic DAG tasks,” in *ECRTS*, July 2015.

[22] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson, “A flexible real-time locking protocol for multiprocessors,” in *RTCSA*, 2007.

[23] R. Davis, A. Burns, J. Marinho, V. Nelis, S. Petters, and M. Bertogna, “Global and partitioned multiprocessor fixed priority scheduling with deferred preemption,” *ACM TECS*, no. 3, May 2015.

[24] R. Vargas, E. Quinones, and A. Marongiu, “OpenMP and timing predictability: a possible union?” in *DATE*, 2015.

[25] J. Marinho, V. Nélis, S. M. Petters, M. Bertogna, and R. I. Davis, “Limited pre-emptive global fixed task priority,” in *RTSS*, Dec. 2013.

[26] *AUTomotive Open System ARchitecture (AUTOSAR) Operating System*, AUTOSAR GbR, http://www.autosar.org. February 2013.

[27] S. Kramer, D. Ziegenbein, and A. Hamann, “Real world automotive benchmark for free,” in *Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS)*, July 2015.

[28] H. Ozaktas, C. Rochange, and P. Sainrat, “Automatic WCET analysis of real-time parallel applications,” in *13th Workshop on Worst-Case Execution Time Analysis (WCET)*, July 2013.

[29] E. Bini and G. C. Buttazzo, “Measuring the performance of schedulability tests,” *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, 2005.