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.
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 $T = \{\tau_1, \ldots, \tau_n\}$ executing according to global fixed-priority scheduling on a platform composed of $m$ identical cores. Each task $\tau_k$ releases an infinite sequence of jobs with a minimum inter-arrival time $T_k$ 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., $\tau_i$ has a higher priority than $\tau_j$ if $i < j$. We denote as $hp(k)$ and $lp(k)$ the subsets of tasks with higher and lower priorities than $\tau_k$, respectively.

Each task $\tau_k \in T$ 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 $\text{len}(G_k)$ 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 $\text{vol}(G_k)$ 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 are 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 $\tau_k$ 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
Assume that tasks $\tau_1$, $\tau_2$, $\tau_3$, and $\tau_4$ on a 2-core processor. For simplicity, the task-set is composed of sequential tasks. Similarly, task $\tau_3$ represents the lower-priority interference of task $\tau_1$ with one or more 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 $\tau_k$ may start executing along with one or more lower priority instances $\tau_{i>0}$. If a higher priority task arrives, $\tau_k$ 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 $\tau_k$ to be de-scheduled, while tasks $\tau_{i>0}$ 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.

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.
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 $\tau_k$ may experience lower-priority blocking even without being preempted by higher priority tasks, e.g., when a node of $\tau_k$ forks two or more parallel nodes, requiring additional cores to execute them. If cores are busy executing lower priority instances, $\tau_k$ 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, $\tau_1$ and $\tau_2$, executing on $m = 2$ cores. The structure of $\tau_1$ and $\tau_2$ is as in Figure 3(a), assuming all nodes have unitary WCET, except $v_{2,1}$ with $C_{2,1} = 2$. Task $\tau_1$ is the highest priority task, therefore it cannot be preempted after it started executing. Still, $\tau_1$ may be blocked by $\tau_2$ once $\tau_1$ 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 $\tau_2$ executing on the other core.

To identify the additional number of priority inversions once a task $\tau_k$ 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 $\tau_k$ may cause after starting its execution.

These are all the points at which a potential priority inversion may arise after $\tau_k$ 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 $\tau_1$, $sw_1 = 1$ 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 $\tau_2$, $sw_2 = 0$ because it does not require extra cores after it started executing. Finally, $sw_3 = 4$ because $\tau_3$ 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 $\tau_3$ occupies three cores when nodes $v_{3,4,5}$ 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 ($v_{3,3}$’s successors minus 1) because $v_{3,4}$ should execute before $v_{3,5}$.

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 $\tau_k$ reaching a preemption
point is preempted. As a result, $\tau_k$ can suffer lower priority interference not only before starting its execution, but also at later preemption points. Moreover, with a DAG-task model, $\tau_k$ 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 $\tau_k$.

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

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

**Proof:** Assume the job of $\tau_k$ is released at time $t_0 = 0$. During a time interval of length $t$, each higher-priority task $\tau_i$ 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 $\tau_k$) by a single instance of $\tau_i$ is at most $1 + sw_i$: one when $\tau_i$ 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 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 $\tau_k$ that already started executing may experience additional lower-priority interference only if all the following conditions are simultaneously satisfied:

1) $\tau_k$ encounters a preemption point.
2) A higher priority task arrives OR $\tau_k$ 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 $\tau_k$ 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

$$p^{eager}_k(t) = \min\left(q_k, sw_k + h_k(t), \sum_{\forall \tau_i \in lp(k)} \left\lceil \frac{t + R_i}{T_i} \right\rceil \times |V_i| \right)$$  \hspace{1cm} (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 $\tau_k$ 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 $t$ ($\sum_{\forall \tau_i \in lp(k)} \left\lceil \frac{t + R_i}{T_i} \right\rceil \times |V_i|$).

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

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

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

$$\Delta_k^m = \sum_{l=1}^{m} Q_l^k \quad \text{and} \quad \Delta_k^{m-1} = \sum_{l=1}^{m-1} Q_l^k$$  \hspace{1cm} (6)

$\Delta_k^m$ upper-bounds the lower priority interference on the first NPR of $\tau_k$, while $\Delta_k^{m-1}$ upper-bounds the lower priority interference on the $p^{th}$ NPRs ($2 \leq p \leq q_k + 1$) of $\tau_k$, i.e. when a priority inversion occurs. $Q_l^k$ denotes the $l^{th}$ largest NPR in the set $lp(k)$. In order to upper bound the lower-priority blocking suffered by a task $\tau_k$, the analysis in [14] considers that $\tau_k$’s lower priority tasks are executing their $m$ longest NPRs on the $m$ available cores when $\tau_k$ is released. Moreover, in the worse case, at each $\tau_k$’s potential priority inversion (due to a higher priority tasks or because $\tau_k$ requires extra cores), the $m$–1 longest NPRs of $\tau_k$’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 $\tau_k$ 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 under-estimates 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 interfere with $\tau_k$ ($\sum_{\forall \tau_i \in lp(k)} \left\lceil \frac{t + R_i}{T_i} \right\rceil \times |V_i|$), 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 $\tau_k$ 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, $\tau_k$ can only be preempted by higher priority tasks if it is the lowest priority task executing on the processor. As a result, $\tau_k$ 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 $\tau_5, \tau_6, \tau_7$ and $\tau_8$ are already executing...
on the processor, when the higher-priority tasks \( \tau_1, \tau_2, \tau_3 \) and \( \tau_4 \) are simultaneously released at time instant \( t_1 \). The first task to be preempted is \( \tau_6 \) (the lowest priority task) at time \( t_2 \), when a preemption point is reached, and so the highest priority ready task \( \tau_1 \) can start its execution. In the worst case scenario task \( \tau_7 \) reaches a preemption point at time \( t_2 + \varepsilon \) in which the lowest priority task \( \tau_7 \) is still executing, and so it continues to execute (an so blocking \( \tau_2 \)) until its next preemption point is reached at \( t_3 \), when \( \tau_2 \) can start executing. Subsequently in the worst-case situation, \( \tau_6 \) enters in a node just before the preemption point of \( \tau_2 \) is reached at time \( t_3 \), blocking \( \tau_3 \) until \( t_4 \). Finally, \( \tau_1 \) is able to start its execution at time \( t_5 \). Overall, the worst-case blocking of \( \tau_2 \) 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 \( lp(\tau) \) multiplied by \( m \), the second largest node from the \( lp(\tau) \) multiplied by \( m-1 \), the third largest node from the \( lp(\tau) \) multiplied by \( m-2 \), and so on, until the \( m \)-th largest nodes from the \( lp(\tau) \) 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 \( \tau_1, \tau_2, \tau_3 \) and \( \tau_4 \), in decreasing priority order (Figure 5(a)), scheduled on \( m = 3 \) cores (Figure 5(b)). We assume that tasks \( \tau_2, \tau_3 \) and \( \tau_4 \) are executing its first node when the highest-priority task \( \tau_1 \) is released at time \( t_1 \). Under the lazy approach, \( \tau_1 \) starts executing the first node \( v_{1,1} \) when the lowest-priority running task \( \tau_4 \) 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 \( t_4 \), \( \tau_2 \) reaches a preemption point, but since it is not the lowest priority task, node \( v_{1,3} \) of \( \tau_1 \) is blocked until \( t_3 \) 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 \( \tau_2 \) and \( \tau_3 \) block the execution of the intermediate node \( v_{1,3} \). The reason is because, at time \( t_4 \), \( \tau_1 \) spawned two parallel nodes, and so requested one extra core.

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

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

\[
\begin{align*}
\text{proof:} & \quad \text{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 } \tau_k \text{ can only be preempted by a higher-priority task and if it is the case, following the lazy approach, } \tau_k \text{ would be the lowest priority tasks (there are not other lower priority instances running and blocking } \tau_k \text{). However, if } \tau_k \text{ requires extra cores to spawn new parallel nodes, all cores may be occupied by lower priority tasks blocking } \tau_k \text{. Condition (3) is trivially derived by noticing that no lower-priority blocking may be experienced without lower-priority instances being executed.} \\
& \quad \text{Lemma 5 enables upper-bounding the maximum number of priority inversions for the lazy approach as follows:} \\
& \quad \text{Lemma 6: With the LP lazy approach, in any time interval of length } t, \text{ an upper bound on the number of additional priority inversions that a DAG-task may experience after starting its execution is} \\
& \quad p_{k,lazy}^{\tau_k}(t) = \min_{\forall \tau_i \in p(k)} \left( \frac{t + R_k}{T_i} \times |V_i| \right) \\
& \quad \text{Proof: Similarly to the eager approach, the number of additional priority inversions is upper bounded by the number of extra cores requests } sw_k \text{ (see Definition 1) and the number of lower priority tasks’ nodes } \left( \sum_{\forall \tau_i \in p(k)} \frac{t + R_k}{T_i} \times |V_i| \right). \\
& \quad \text{Despite Lemma 5 defines as a necessary condition to encounter a preemption point, } q_k > sw_k \text{ 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 \text{ different nodes. In this case, } q_k = m \text{ while } sw_k = m - 1 \text{ as the same core executing the first node can execute one of the spawned nodes.} \\
& \quad \text{Overall, and following the approach in [14], the lower priority interference } I_{k,lazy}^p, \text{ for the LP lazy approach is computed as:} \\
& \quad I_{k,lazy}^p = A_k^m + p_{k,lazy}^{\tau_k}(R_k) \times A_k^{m-1} \\
& \quad \text{where (considering the blocking estimation presented in [25], named ADS blocking estimation 2)}: \\
& \quad 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 \text{As shown in the previous Section VI, } Q_k^l \text{ denotes the } l^{th} \text{ largest NPR in the set } lp(k). \\
\end{align*}
\]

**VIII. Evaluation**

This section evaluates the proposed response time analysis of both the eager and lazy limited preemption approaches.
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 $R_i, \forall r_i \in lp(k)$, is not computed yet when, iterating over task $k$, we need to compute $p^e_{k,r}$ or $p^l_{k,r}$ (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. $I^B_{k,r}$ 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 $p_{par} = 6$; 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.

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. $p^e_{k,r}$ (Equation 4) and $p^l_{k,r}$ (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.

![Figure 6. Number of additional priority inversions of LP-eager ($p^e_{k,r}$) and LP-lazy ($p^l_{k,r}$) and DAG's maximum number of preemptions ($q_k$), when varying the number of tasks.](image-url)
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, $p_{eager}^k$ is given by $q_k$ except for (1) the highest priority task, for which $p_{eager}^k = sw_k$ because $b_k = 0$ and $sw_k < q_k$, and (2) the lowest priority task, for which $p_{eager}^k = 0$ 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 $p_{eager}^k$. In most cases, $p_{lazy}^k$ is given by $sw_k$, except for the lowest priority task, for which $p_{lazy}^k = 0$ 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 $10^9$ 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. Concretely, the Figure shows the contribution of $I_{lpazy}^k$ and $I_{eager}^k$, 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 $I_{lpazy}^k$ alone is also shown for FP-ideal.

As shown, the blocking factor due to lower priority tasks of LP-lazy ($I_{lpazy}^k$) 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, $I_{lpazy}^k$ 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 $I_{lpazy}^k$ or $I_{eager}^k$ increase, the window of interest increases as well, impacting on $I_{hp}^k$.

Overall, factors $A_k^m$ and $A_k^{m-1}$ (used in $I_{lpazy}^k$) add huge pessimism, increasing the window of interest and impacting on the system schedulability of LP-lazy as shown in next section.
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 $I^L_k$ has been computed using Equations 5 ($I^L_{eager}$) and 8 ($I^L_{lazy}$) 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 $U_{Unifast}$ [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, $I^L_{lazy}$ 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
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