Job-shifting: An algorithm for online admission of non-preemptive aperiodic tasks in safety critical systems

Ali Syed\textsuperscript{a,1,\ast}, Daniel Gracia Pérez\textsuperscript{b}, Gerhard Fohler\textsuperscript{a}

\textsuperscript{a} Technische Universität Kaiserslautern, Kaiserslautern, Germany
\textsuperscript{b} Thales Research & Technology, France

ARTICLE INFO

Keywords:
Real-time systems
Scheduling algorithms
Mixed-criticality
Hierarchic scheduling
Online aperiodic admission

ABSTRACT

Modern safety critical systems require certification in order to guarantee correct operation before system deployment. The certification process requires rigorous verification and validation, the efforts for which can be greatly reduced by using resource partitioning. However, Lackorzyński et al. demonstrated that bandwidth reservation for event-triggered (ET) activities in partitioned systems may lead to significant bandwidth loss. In contrast, the online admission of ET activities can prevent bandwidth losses. However, the state-of-the-art approaches for online admission of ET activities fail to fulfill the requirements of safety critical systems as they do not support (i) partitioning, (ii) the industrial mixed-criticality task model or (iii) non-preemptive task execution. In this paper, we present job-shifting algorithm for online admission of non-preemptive aperiodic tasks in partitioned time-triggered environment. Our approach circumvents the bandwidth loss issue with partitioning, and provides guarantees similar to the bandwidth reservation technique such that the certification process of safety critical systems need not be modified. Our approach can be implemented on top of variety of hypervisors and can provide lower response-times for aperiodic tasks. Through evaluation, we demonstrate that our approach efficiently utilizes processor bandwidth and only incurs small scheduling overheads.

1. Introduction

For safety critical applications, the certification process makes sure that the system will meet the required constraints upon deployment. This process is utilized to define if the product can be safely deployed for the desired application. However, the certification process requires rigorous verification and validation (V&V) of the system, and therefore, the process complexity increases as the system size increases. Partitioning or virtualization enables certification of a sub-system irrespective of the behaviour of other sub-systems, and as a result, reduces the V&V efforts \cite{1}.

Depending on the application requirements, a computing system may be required to handle several constraints. In real-time systems, periodic tasks can easily be serviced using time-triggered (TT) mechanisms due to predictable repetitive arrival patterns. However in the case of aperiodic tasks, TT mechanisms like offline bandwidth reservation may lead to over-provisioning, and therefore, may increase cost per feature. The over-provisioning problem becomes increasingly prohibitive with virtualization technologies as these technologies may lead to significant bandwidth loss \cite{2}. Therefore, a new algorithm needs to be developed which provides efficient integration of periodic and aperiodic activities in hierarchic, industrial mixed-critical environments.

Besides bandwidth reservation, a number of techniques are proposed over the last few decades to integrate time-triggered (TT) and event-triggered (ET) activities, for instance, slack-stealing \cite{3} and slot-shifting \cite{4}. The basic operating principle for both of these approaches is to estimate the information about the amount and distribution of free resources offline and to accommodate aperiodic tasks online. A number of extensions were also proposed for both of these algorithms to compensate for new constraints, e.g., sporadic tasks \cite{5}, non-preemptive aperiodic tasks in a preemptive scheduling environment \cite{6} and Vestal’s \cite{7} mixed-criticality task model \cite{8}. Although these extensions are viable solutions, they need to frequently keep track of free resources, they are computationally expensive and they cannot be applied on partitioned systems.

In this paper, we present an algorithm which defines availability of free resources in an offline TT scheduling table of non-preemptive tasks, and employs the availability information online to service non-preemptive aperiodic tasks in a partitioned system; we call this algorithm...
‘job-shifting’. Although non-preemptive scheduling offers several benefits (e.g. reduced I/O delays [9], precision in the estimation of WCET [10], etc.), it is an NP-Hard problem [11]. Consequently, the non-preemptive versions of EDF (e.g. npEDF) are not optimal [9]. Therefore, the job-shifting algorithm is not designed to confine the application integrator to the EDF strategy (unlike slot-shifting [4,6,8]) and can utilize either online or offline scheduling strategies. The job-shifting algorithm provides aperiodic task execution guarantees similar to the bandwidth reservation techniques, which means that there is no need to modify the certification process of the safety critical application. Furthermore, the job-shifting algorithm provides control over task jitter, can be used with sparse and dense time-bases [12] and does not require computationally expensive slot-based record keeping mechanism.

We highlight the advantages and disadvantages of the job-shifting algorithm and provide a necessary condition for online aperiodic admission which can be checked offline. Furthermore, we also identify which factors affect the overheads and complexity of the algorithm implementation. Through simulation, we demonstrate that the proposed algorithm efficiently utilizes free resources in hierarchical schedules. Moreover, we evaluate the overheads introduced by the job-shifting algorithm on the Zyqng ZC706 board2, and demonstrate that the incurred overheads (i) are small and (ii) are comparable to the overheads incurred by the state-of-the-art preemptive approaches.

This paper is a revision of our previous work [13] which we extended as follows: We provide (i) proofs to support the claim that job-shifting utilizes every opportunity to admit aperiodic tasks, (ii) extension of the job-shifting algorithm to support execution of both critical and non-critical tasks in a single partition, (iii) an offline strategy for the management of free resources to exploit processor when no partition is executed, (iv) the memory requirements of the job-shifting algorithm, (v) the strategy to control task jitter when using job-shifting algorithm, (vi) extended efficiency evaluation to account for varying utilization of periodic tasks inside the partition, (vii) details of the job-shifting integration strategy in the DREAMS3 tool-chain and (viii) job-shifting overheads evaluation with schedule implementation as a double linked list.

The reminder of this paper is organized as follows: Section 2 presents an overview of the related work. Section 3 introduces the system model utilized by the job-shifting algorithm. Section 4 provides a detailed description of the job-shifting algorithm, and highlights the suitability of the algorithm in safety-critical system. Section 5 provides the description of the experimentation for the job-shifting efficiency evaluation. Section 6 presents the overhead evaluation experiment with description of the safety critical demonstrator and the job-shifting implementation in the DREAMS project. Section 7 discusses in detail how to improve the performance of the job-shifting algorithm. And finally, Section 8 concludes the paper and provides pointers for the future work.

2. Related work

A trivial approach to service aperiodic tasks is to use interrupt mechanism. However with this approach, it is difficult to provide guarantees due to complex interference patterns [14] and the methodology is strongly dependent on the hardware platform. A better strategy is to use background processing to service aperiodic tasks. However, background processing provides no guarantees for aperiodic task execution and leads to large aperiodic response times.

Over the last few decades, a strong theoretical background has been established for the bandwidth reservation technique (aka aperiodic servers). The technique is employed in safety critical systems to service sporadic and aperiodic tasks [15] due to ease of implementation and less V&V efforts. In the bandwidth reservation technique, the aperiodic or sporadic tasks are transformed to periodic tasks by defining the parameters period and budget. Although it is debatable how these parameters can be defined for an aperiodic activity, the values for these parameters are selected based on educated-guesses or best-practices. Moreover, the bandwidth reservation technique leads to bandwidth loss when (i) partitioning [16] is used [2] and when (ii) the aperiodic activation frequency is less compared to the estimated frequency. Furthermore, the aperiodic response time with this technique strongly depends on the server type, server priority and scheduling strategy (i.e. fixed- or dynamic-priority scheduling).

As the approaches mentioned above present several bottle-necks, from this point on we will focus on the online admission paradigm for aperiodic task execution. As there exist a significant number of algorithms which utilize online admission approach (e.g. [17]), we will only discuss two major ones (i.e. slack-stealing [3] and slot-shifting [4]) due to their support for variety of system models.

In the early 90s, Lehoczky et al. defined slack-stealing algorithm to service soft [3] and hard [18] aperiodic task in fully-preemptive fixed-priority systems. In this algorithm, they precomputed and stored the slack function of the task-set and used it online to service aperiodic tasks. The slack-stealing algorithm was later extended by Tia et al. [19] to efficiently compute the slack function online, since the size of the slack function table may be too large, depending on the periods of the tasks.

Similarly, Fohler [4] defined slot-shifting algorithm for aperiodic admission in fully-preemptive TT dynamic-priority systems (specifically EDF) with sparse time-base [12]. In this algorithm, capacity (or slack) intervals and their spare capacities are calculated offline and stored in a table to be used online to service soft and aperiodic and sporadic tasks [5]. The slot-shifting algorithm requires a significant amount of memory and utilizes an online mechanism to keep track of used resources activated at each slot boundary [12].

Recently, Schorr [6] extended the original slot-shifting algorithm to allow admission of non-preemptive aperiodic tasks in a preemptive schedule. The extension utilized the original precomputed offline scheduling table defined by slot-shifting [4], however the aperiodic admission test (termed acceptance test [4]) for non-preemptive aperiodic tasks was modified to calculate enough consecutive slots to execute the released aperiodic job. Moreover, the guarantee algorithm [4] was modified to improve the response-time at the cost of flexibility [6]. Similar to the original slot-shifting algorithm, the extension by Schorr requires a significant amount of memory and uses an online slot based record keeping mechanism. Moreover, the slot-shifting extension proposed by Schorr has higher complexity and larger run-time overheads.

Similarly, Theis [8] extended the slot-shifting algorithm to support mixed-critical tasks (based on Vestal’s mixed-criticality task model [7]) and mode changes. Similar to the original slot-shifting algorithm, the extension by Theis is based on EDF, however the memory requirements (and therefore the overheads) are increased proportional to the number of modes or criticalities.

In this paper, we present job-shifting algorithm, which can be used in industrial hierarchical mixed-critical systems (not based on Vestal’s mixed-criticality task model [7]) to admit non-preemptive aperiodic tasks. Unlike the slot-shifting algorithm [4,6,8], the job-shifting algorithm does not require (i) slot-based record keeping mechanism, (ii) sparse time-base [12] or (iii) EDF scheduling. The job-shifting algorithm respects separation of concerns (through partitioning), incurs small scheduling overheads compared to the state-of-the-art and provides better response-times for aperiodic activities. The job-shifting algorithm also provides offline guarantees and control over task jitter. However, job-shifting only supports non-preemptive scheduling, which is NP-Hard even for the simple case of independent tasks with implicit deadlines [11].
3. System model

A partitioning kernel or hypervisor is used to provide strict temporal and spatial isolation ([20–22]), which enables independence of safety functions between applications. In order to fulfill strict safety requirements, cyclic-executive scheduling [16] is assumed to be the inter-partition scheduling strategy. We assume a uni-processor non-preemptive execution environment. Moreover, all activities in the system are assumed to be triggered by the passage of time, i.e. time-triggered with sparse or dense time-base [12]. For digital computing systems, the sparse time-base is implemented by the scheduler using a scheduling quantum significantly larger than the processor clock cycle length, e.g., a slot in slot-shifting [4]. However, when the scheduler does not enforce the sparse time-base, the system is subjected to dense time-base (aka ‘as fast as possible’) where the scheduling quantum is equal to the processor clock cycle length.

In order to apply job-shifting to a partition \( p_i \) in a partition set \( P \), it is assumed that the intra-partition scheduling employs TT scheduling tables with a simple online dispatcher (relaxed in Section 7.5), and contains both periodic and aperiodic tasks. The idle time inside a partition \( p_i \) is also assumed to be non-zero (discussed further in Section 7.8). The time when the partition \( p_i \) is not available to service applications is defined by the blocking set \( V_i \). Each blocking \( v \in V_i \) is defined by the tuple \( [b, m] \), where \( b \) and \( m \) represent the absolute beginning and termination time of blocking \( v \), respectively.

A task-set \( \Gamma \) is defined as a collection of tasks. Each task phase, \( \phi \), of \( \Gamma \) is defined by the tuple \( \langle \phi, C, T, D, Y \rangle \), where \( \phi \) represents the task phase, and \( C \) represents the worst-case execution time (WCET) of the task \( \phi \). When the task is periodic, the parameter \( T \) defines its period; otherwise, \( T = \infty \). Moreover, \( D \) defines the relative constrained deadline (i.e. \( D \leq T \)), and \( Y \) represents the task criticality (i.e. the safety level, further in Section 7.1). An aperiodic task release is usually detected by checking a peripheral specific register, e.g. when a button is pressed, a message is received on a (virtual) network port. A task \( r \in \Gamma \) consists of infinite jobs \( j \), each of which is defined by the tuple \( [r, d, x] \), where \( r \) represents the absolute job release time, and \( d \) defines the absolute job deadline. The jobs \( j \) are assumed non-preemptive, however they can be paused at the partition boundary.

The scheduling table \( S_p \) for a partition \( p \) is constructed prior to the job-shifting offline phase (discussed in Section 4.1) for the length of the scheduling cycle (SC). The length of SC is defined by the following equation [23];

\[
SC = \begin{cases} 
0, \text{LCM} & \forall \tau: \phi = 0 \\
0, \phi_{\max} + 2 \times \text{LCM} & \text{otherwise}
\end{cases}
\]

(1)

where, \( \phi_{\max} \) represents the maximum phase \( \phi \) of all tasks \( r \in \Gamma \), while \( \text{LCM} \) represents the least common multiple of all the periods \( T \) of periodic tasks. For each job in SC, the scheduling table \( S_p \) defines the absolute job activation time \( a \) and the absolute job finish time \( f \). It is assumed that the partition scheduling table \( S_p \) available as input to the job-shifting algorithm, is a valid feasible schedule. The tasks \( r \in \Gamma \) may have precedence and/or mutual exclusion constraints. However, these constraints are assumed to be resolved either by modifying the job release time \( r \) and deadline \( d \) or by constructing the scheduling table \( S_p \).

An example scheduling table for a periodic task \( r_i \) with blocking \( v_0 \) is shown in Fig. 1, while a summary of the used notations is provided in Table 1.

4. Job-shifting algorithm

In this section, we provide details of the job-shifting algorithm to admit non-preemptive aperiodic tasks in flat and hierarchical scheduling models. Without loss of generality, in this section we assume that the sparse time-base [12] is used. Moreover, all the jobs are assumed to execute for the complete worst-case execution time \( C \) every time (relaxed in Section 7.4). Furthermore, as the task criticality \( Y \) denotes the safety standard and not the importance [1], its use will only be discussed in Section 7.1.

4.1. Methodology

In order to apply the job-shifting algorithm, a feasible offline scheduling table \( S_p \) for partition \( p \) is required. To construct \( S_p \), the tasks \( r \in \Gamma \) are allocated to the processing nodes and the partitions \( P \). The periodic tasks are then unrolled to create jobs for the length of SC and the scheduling table \( S_p \) is constructed utilizing the desired scheduler such that the feasibility is ensured. For the offline scheduled partitions, this necessarily means that the values for \( a_i \) and \( f_i \) are defined for each job \( j_i \) inside the partition such that \( \forall i \in S_p: a_i < d_i \land a_i < f_i \leq d_i \land f_i - a_i \geq C_i \).

Once the offline scheduling table \( S_p \) for a partition is ready, the job-shifting algorithm can be applied, which is divided in two phases; an offline phase and an online phase. In the offline phase, a parameter, we call the flexibility coefficient \( x_i \) for each job \( j_i \in S_p \) is calculated (for example, see \( x_0 \) in Fig. 1). We define the flexibility coefficient \( x_i \) as:

**Definition 4.1.** The flexibility coefficient \( x_i \) for job \( j_i \in S_p \) defines the maximum delay, which can be added to the absolute activation time \( a_i \) of job \( j_i \) without changing the execution order of jobs in \( S_p \), and without missing any deadline.

During the online phase of job-shifting algorithm, the scheduler checks the arrival of the aperiodic job(s). When a new aperiodic job is detected, the guarantee test is performed. If the guarantee test succeeds, the new aperiodic job is adjusted in the partition schedule by invoking the guarantee procedure. However, if the guarantee test fails, the aperiodic job is added to the best-effort queue. Upon each activation of the scheduler, a job on the best-effort queue can be executed if there exists enough aperiodic room \( R_i \). We define the aperiodic room \( R_i \) as follows:

**Definition 4.2.** The aperiodic room \( R_i \) defines the maximum contiguous processing node time, which can be spared for executing aperiodic job(s) prior to the activation of job \( j_i \), without missing any deadline in the system.

On account of the aperiodic room definition, it can be observed that the best-effort queue is not a background queue. Instead, a job on the best-effort queue is executed as soon as enough aperiodic room \( R_i \) can be reserved (further in Section 4.4).

4.2. Flat scheduling model

In this section, we assume that there exists only one partition which is available to service tasks at all times, i.e. \( V_p = \emptyset \). Furthermore, the finish time \( f_i \) for a job \( j_i \) is not defined by the scheduling table \( S_p \), instead \( f_i \) is directly calculated by the equation \( f_i = a_i + C_i \).

4.2.1. Offline phase

As mentioned earlier, during the offline phase, the flexibility coefficient \( x_i \) for each job \( j_i \in S_p \) is calculated starting from the job \( j_i \) with the maximum activation time \( a_i \) and ending at the job \( j_i \) with the minimum activation time \( a_i \). Assuming \( a_{i+1} = d_i \), and \( x_{i+1} = 0 \), for each job \( j_i \), the following steps are performed;

i) Calculate the parameter \( O_0 \), which defines the overlap between the flexibility windows \( [a_i, d_i] \) of job \( j_i \) and job \( j_{i+1} \), using the following equation;

\[
O_0 = \max(0, d_i - a_{i+1})
\]

(2)

ii) Calculate the flexibility coefficient \( x_i \) using the following equation;

\[
x_i = d_i - a_i - C_i - O_i + \min(x_{i+1}, O_i)
\]

(3)
In order to clarify that Eq. (3) provides the correct job flexibility as per Definition 4.1, we provide the following lemma.

**Lemma 4.1.** The Eqs. (2) and (3) provide the maximum value by which the job activation time \( a_i \) can be delayed without changing job execution order and without leading to a deadline miss.

**Proof.** Assume two consecutive jobs with activation times \( a_i \) and deadlines \( d_i \) as shown in Fig. 2. Based on the occurrence of the deadline of job \( j_i \), three different intervals \( I_1, I_2 \) and \( I_3 \) can be identified.

When the deadline \( d_i \) lies during the interval \( I_1 \), the flexibility \( x_i \) of job \( j_i \) does not depend on \( j_j \) and therefore, the maximum delay is equal to the job slack [24]. Due to a delay in \( a_i \) equal to the job slack, neither job \( j_i \) nor \( j_j \) will result in a deadline miss.

When the deadline \( d_i \) lies during the interval \( I_2 \), the job overlap \( O_i \) falls in the range \([0, x_i]\). Due to non-zero overlap, the maximum delay in \( a_i \) gets a dependency on \( j_j \) and therefore, the overlap need to be considered. In Fig. 2, the maximum processing time which can be borrowed by \( j_j \) from the execution window of \( j_i \) is given by min \((O_i, x_i)\). If more processing time is borrowed, the job \( j_j \) will lead to a deadline miss. In such a situation, executing \( j_j \) before \( j_i \) may or may not lead to a deadline miss, however, in such a case, the execution order of the schedule is violated.

For the case of interval \( I_3 \), the overlap \( O_i \) becomes larger than \( x_i \) (since the maximum flexibility of a job is given by the job slack [24]). Therefore, the maximum processing time borrowed by \( j_j \) from \( j_i \) is given by \( x_i \). Executing \( j_j \) before \( j_i \) may or may not lead to a deadline miss, however, the execution order of the jobs is violated.

Note that the Eq. (3) gives the maximum delay in all of these cases and, therefore, the lemma is proven.

**Example 4.1.** Assume two jobs \( j_i \) and \( j_j \) with release time \( r \), activation time \( a \), deadline \( d \) and worst-case execution time \( C \) as shown in Fig. 3 and defined in Table 2. During the offline phase of job-shifting algorithm, the flexibility coefficient \( x \) is calculated starting from job \( j_1 \) and ending at job \( j_n \). The calculation is provided below;

For job \( j_i \), \( a_i = d_i \) and \( x_i = 0 \).

\[
O_i = \max(0, d_i - a_i) = 0
\]

\[
x_i = d_i - a_i - C_i - O_i + \min(x_i, O_i)
\]

\[
= 10 - 2 - 2 - 0 + \min(0, 0)
\]

\[
= 6
\]

For job \( j_n \), \( a_2 = 2 \) and \( x_2 = 6 \).

\[
O_n = \max(0, d_n - a_n) = \max(0, 8 - 2) = 6
\]

\[
x_n = d_n - a_n - C_n - O_n + \min(x_n, O_n)
\]

\[
= 8 - 0 - 2 - 6 + \min(6, 6)
\]

\[
= 6
\]

**4.2.2. Online phase**

After finishing the offline phase, all the parameters for each job are passed to the online scheduler, which is activated at each job activation time \( a_i \) finish time \( f_i \) and (when idle) at the release of a new aperiodic job.

The guarantee test is performed when the scheduler gets activated at time \( t \) and an aperiodic job release is detected. For the guarantee test, the job \( j_h \) is defined as the next job to be activated from the scheduling table \( S_p \), while the job \( j_i \) is defined as the first job activated after the deadline of the released aperiodic job \( j_{lp} \). During the guarantee test, the aperiodic room \( R_i \) for each job \( i \) from \( j_h \) to \( j_i \) is calculated using the following set of equations:

\[
s_i = \begin{cases} a_{i-l} + C_{l-l}, & n < i < l \\ t_i, & i = n \end{cases}
\]

\[
R_i = a_i - s_i + \min(d_{lp} - a_i, x_i)
\]

**Table 2**

Example parameters for offline phase of flat scheduling model.

<table>
<thead>
<tr>
<th>( i )</th>
<th>( r )</th>
<th>( a )</th>
<th>( d )</th>
<th>( C )</th>
<th>( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>10</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>
where, $s_i$ represents the (potential) start of the released aperiodic job $j_{ap}$ before job $j_i$. In order to guarantee that the equation system (4) provides the correct aperiodic job room as per Definition 4.2, we provide the following lemma.

**Lemma 4.2.** The equation system (4) provides the maximum processing time $R_i$ available for the execution of an aperiodic job before job $j_i$ without leading to a deadline miss.

**Proof.** Assume a set of guaranteed jobs $j_1, j_{a+1}, ..., j_n$, with activation times $a$ and deadlines $d$ as shown in Fig. 4. When an aperiodic job $j_{ap}$ is released at time $t_{ap}$ with a deadline $d_{ap}$ and worst-case execution time $C_{ap}$, three different intervals $I_1, I_2$ and $I_3$ can be identified.

When the deadline $d_{ap}$ lies during the interval $I_1$, the maximum aperiodic room $R_0$ for job $j_i$ will be $d_{ap} - t_{ap}$. Executing an aperiodic job with $C_{ap}$ larger than $R_0$ may result in a deadline miss by job $j_{ap}$. Note that in this case, the difference $d_{ap} - a_i$ results in a negative value.

When the deadline $d_{ap}$ lies during the interval $I_2$, the maximum processing time which can be borrowed by $j_{ap}$ from the execution window of $j_i$ is given by $\min(O_{ap}, x_0)$. If more processing time is borrowed, the job $j_i$ will miss the deadline.

For the case of interval $I_3$, the overlap $O_{ap}$ becomes larger than $x_0$ (see Lemma 4.1). Therefore, the maximum processing time borrowed by $j_{ap}$ from $j_i$ is given by $x_0$. Executing a job with $C_{ap}$ greater than $a_i - t_{ap} + x_0$ might result in a deadline miss by job $j_{ap}$.

Note that the same reasoning can be applied for all jobs $j_{a+1}, j_{a+2}, ..., j_{n-1}$, however for these jobs, the maximum aperiodic room $R$ will be defined as $a_{a+1} - f_n$ for the first case. Note that the equation system (4) gives the maximum room in all of these cases and, therefore, proves the lemma. □

The guarantee test passes when, for any job $j_{1} | n \leq i < l$, the aperiodic room $R_i$ is larger than or equal to $C_{ap}$. If the guarantee test passes, the aperiodic job $j_{ap}$ can be guaranteed to finish before its deadline $d_{ap}$ without missing any other deadline in the system. Once the guarantee test passes, the guarantee procedure is activated. The guarantee procedure requires the following steps to be performed in the defined order:

1. Insert the released aperiodic job $j_{ap}$ in the schedule $S_p$ before job $j_i$, i.e., $a_i = a_{ap} = s_{ap}$.
2. For each job $j_k$, such that $k > i \land a'_k > a_k$, perform the following steps starting from $j_{k+1}$:
   (a) $a'_k = a_k + \max(a_{k+1} + C_{k+1} - a_k, 0)$
   (b) $x_k = x_k - \left( a'_k - a_k \right)$
   (c) $a_k = a'_k$
3. For each job $j_p$, such that $p = i$, calculate $x_p$ similar to the off-line phase starting from $j_i$ and ending at $j_p$. The process of modifying $x_p$ stops when the old $x_p$ is equal to the new $x_p$.

At the scheduler activation time $t$, the scheduling decision can be made after handling the released aperiodic job(s). If the best-effort queue is empty, the next job with $a = i$ is scheduled from $S_p$. When there exists no such job, the processor is left idle. If there exists a job $j_i$ in the best-effort queue and the next job aperiodic room $R_n \geq C_n$, the guarantee procedure is performed for job $j_i$ with $a_i = t$ and it is scheduled.

**Example 4.2.** Assume an aperiodic job $j_{ap}$ with $C_{ap} = 2$ and $d_{ap} = 3$ is released at time $t = 0$ in the example shown in Fig. 5. The online scheduler performs the guarantee procedure with $n = 1$ and $l = 3$ (i.e., first job of the next SC). For job $j_i$ with $i = 1, s_1 = t = 0$. The aperiodic room $R_1$ is calculated as under;

$$R_1 = a_1 - x_1 = \min(d_{ap} - a_1, x_1)$$

$$= 0 - 0 + \min(3 - 0, 6)$$

$$= 3$$

Since $R_1 > C_{ap}$, the newly released job $j_{ap}$ can be guaranteed to finish before its deadline $d_{ap}$. Therefore, guarantee procedure is triggered. The steps are shown below;

1. Job $j_{ap}$ is inserted at location 0, i.e. $a_0 = a_{ap} = s_1 = 0$. $C_0 = C_{ap} = 2$ and $d_0 = d_{ap} = 3$.
2. For job $j_1$:

   $$a'_1 = a_1 + \max(a_0 + C_0 - a_1, 0)$$

   $$= 0 + \max(0 + 2 - 0, 0) = 2$$

   $$x_1 = x_1 - (a'_1 - a_1) = 6 - (2 - 0) = 4$$

   $$a_1 = a'_1 = 2$$

3. For job $j_2$:

   $$a'_2 = a_2 + \max(a_1 + C_1 - a_2, 0)$$

   $$= 2 + \max(2 + 2 - 2, 0) = 4$$

   $$x_2 = x_2 - (a'_2 - a_2) = 6 - (4 - 2) = 4$$

   $$a_2 = a'_2 = 4$$

The modified scheduling table is shown in Fig. 5, while the modified parameters are highlighted in Table 3. After performing the guarantee test, the job $j_l$ is selected for execution since $a_0 = t = 0$ and the best-effort queue is empty. When the job $j_l$ completes, the scheduler gets activated again at $t = 2$ and schedules $j_1$ and later, at $t = 4$, $j_2$ to complete the execution of the scheduling table.

**4.3. Hierarchical scheduling model**

In this section, we assume that there exist multiple partitions in the system and the job-shifting algorithm is enabled in partition $p \in P$, i.e. $V_p = \emptyset$. For hierarchical scheduling model, we need to define two operators; the Blocking operator $B(p, q)$ and the Adjust operator $A(u)$. The blocking operator $B(p, q)$ returns the sum of the partition blocking duration between the interval $[p, q]$. The operator $A(u)$ modifies the time instant $u$ such that it does not lie within the partition blocking time.
When the time instant $u$ denotes the job finishing time $f_i$, the operator $A(u)$ also makes sure that the duration between the job activation time $a_i$ and the job finish time $f_i$ is enough to complete the job, i.e. $f_i = a_i + B(a_i, f_i) + C$. Unlike the flat scheduling model, the finish time $f_i$ for a job $i$ in the hierarchical scheduling model is also defined by the scheduling table $S_p$. Moreover, it is assumed that the parameters $a_i$ and $f_i$ are adjusted offline using the adjust operator $A(u)$.

### 4.3.1. Offline phase

Similar to the offline phase of flat scheduling model, the flexibility coefficient $x_i$ for each job $i \in S_p$ is calculated during the offline phase of hierarchical scheduling model starting from the job $i$ with the maximum activation time $a_i$ and ending at the job $i$ with the minimum activation time $a_i$. Assuming $a_{i+1} = d_i$ and $x_{i+1} = 0$, for each job $i$ the following steps are performed:

1. Calculate the overlap parameter $O_i$ using the following equation:
   \[
   O_i = \max(0, d_i - a_i + B(a_i, d_i))
   \]  
   \[ (5) \]

2. Calculate the flexibility coefficient $x_i$ using the following equation:
   \[
   x_i = d_i - a_i - C_i - B(a_i, d_i) - O_i + \min(x_{i+1}, O_i)
   \]  
   \[ (6) \]

The validity of Eq. (6) can be proven similarly to Lemma 4.1.

#### Example 4.3.3

Assume two jobs $j_1$ and $j_2$ with release time $r$, activation time $a$, finish time $f$, deadline $d$ and worst-case execution time $C$ as shown in Fig. 6 and defined in Table 4. Also assume that the partition blocking set $V$ is defined to be $(S_1, S_2)$. During the offline phase of job-shifting algorithm, the flexibility coefficient $x_i$ is calculated starting from job $j_2$ and ending at job $j_1$. The calculation is provided as below,

For job $j_2$, $a_3 = d_2$ and $x_3 = 0$,

\[
O_3 = \max(0, d_2 - a_3 - B(a_3, d_2)) = 0
\]

\[
x_1 = d_2 - a_2 - C_2 - B(a_2, d_2) - O_3 + \min(x_3, O_3)
= 10 - 2 - 3 - 6 + \min(0, 0)
= 3
\]

For job $j_1$, $a_2 = 2$ and $x_3 = 0$,

\[
O_1 = \max(0, d_1 - a_2 - B(a_2, d_1))
\]

\[
x_1 = d_1 - a_1 - C_1 - B(a_1, d_1) - O_1 + \min(x_2, O_1)
= 8 - 0 - 2 - 3 - 6 + \min(6, 6)
= 3
\]

![Fig. 6. Example for offline phase of hierarchic scheduling model.](image)

#### 4.3.2. Online phase

After finishing the offline phase, all the parameters for each job and the partition blockings $V_p$ are passed to the online scheduler. The online scheduler is activated at each job activation time $a_i$, finish time $f_i$, and when the partition is active and the processor is idle at the release of a new aperiodic job.

When the scheduler is activated at time $t$ and an aperiodic job release is detected, the guarantee test is performed. Similar to the flat scheduling model, the job $i_p$ is defined as the next job to be activated from the scheduling table $S_p$, while the job $i$ is defined as the first job activated after the deadline of the released aperiodic job $i_p$. During the guarantee test, the aperiodic room $R_i$ for each job $i$ from $i_p$ to $i$ is calculated using the following set of equations:

\[
s_i = \begin{cases} f_{i-1} - n < i < l \\ t_i & i = n \end{cases}
\]

\[
R_i = a_i - s_i - B(s_i, a_i) + \min(d_{i_p} - a_i - B(a_i, d_{i_p}), s_i)
\]

\[ (7) \]

The validity of Eq. (7) can be proven similarly to Lemma 4.2.

The guarantee test passes if, for any job $i$ such that $i < i_p$, the aperiodic room $R_i$ is larger than or equal to $C_{ap}$. Once the guarantee test passes, the guarantee procedure is activated. The guarantee procedure requires the following steps to be performed in the defined order:

1. Insert the released aperiodic job $i_p$ in the schedule $S_p$ before job $i$, i.e. $a_i = a_{i_p} = s_{i_p} + 1$ and $f_i = f_{i_p} = A(a_{i_p} + C_{ap})$.
2. For each job $i_p$, such that $k > i$ and $a_k > a_i$, perform the following steps starting from $i_p$:
   (a) $a_k' = A(a_k + \max(f_{i_p-1} - a_k, 0))$
   (b) $f_k' = A(f_k + (a_k' - a_k - B(a_k, a_k'))$
   (c) $x_k = x_k - (a_k' - a_k - B(a_k, a_k'))$
   (d) $a_k = a_k'$
3. For each job $i_p$, such that $p < i$, calculate $x_p$ similar to the offline phase starting from $i_p$ and ending at $i$. The process of modifying $x_p$ stops when the old $x_p$ is equal to the new $x_p$.

During run-time, the jobs are scheduled similar to the flat scheduling model.

#### Example 4.4.4

Assume an aperiodic job $i_p$ with $C_{ap} = 2$ and $d_{i_p} = 3$ is released at time $t = 0$ in the example shown in Fig. 6. The online scheduler performs the guarantee procedure with $n = 1$ and $l = 3$ (i.e. first job of the next SC). For job $i_p$ with $i = 1$, $s_i = t = 0$. The aperiodic room $R_i$ is calculated as under,

\[
R_i = a_i - s_i + \min(d_{i_p} - a_i - B(a_i, d_{i_p}), s_i)
\]

\[
= 0 - 0 + \min(3 - 0 - 0, 3) - 0
\]

\[
= 3
\]

Since $R_i > C_{ap}$, the newly released job $i_p$ can be guaranteed to finish before its deadline $d_{i_p}$. Therefore, guarantee procedure is triggered. The steps are shown below:

1. Job $i_p$ is inserted at location $0$, i.e. $a_0 = d_{i_p} = s_0 = 0$,
   
   \[
f_i = f_{i_p} = A(a_{i_p} + C_{ap}) = 2, C_0 = C_{ap} = 2\text{ and }d_0 = d_{i_p} = 3.
\]
2. For job $j_1$:
\[
a_1' = A(a_1 + \max(f_0' - a_1, 0)) \\
= A(0 + \max(2 - 2, 0)) = 2 \\
f_1' = A(f_0' + a_1' - a_1 - B(a_1, a_1')) \\
= A(4 + (4 - 2 - B(2, 4))) = 9 \\
x_1 = x_1 - (a_1' - a_1 - B(a_1, a_1')) \\
= 3 - (4 - 2 - B(2, 4)) = 1 \\
a_1 = a_1' = 2
\]

For job $j_2$:
\[
a_2' = A(a_2 + \max(f_0' - a_2, 0)) \\
= A(2 + \max(4 - 2, 0)) = 4 \\
f_2 = A(f_0' + a_2' - a_2 - B(a_2, a_2')) \\
= A(4 + (4 - 2 - B(2, 4))) = 9 \\
x_2 = x_2 - (a_2' - a_2 - B(a_2, a_2')) \\
= 3 - (4 - 2 - B(2, 4)) = 1 \\
a_2 = a_2' = 4
\]

3. For job $j_0$:
\[
O_0 = \max(0, d_0 - a_1 - B(a_0, d_0)) \\
= \max(0, 3 - 2 - B(2, 3)) = 1 \\
x_0 = d_0 - a_0 - C_0 - B(a_0, d_0) - O_0 \\
+ \min(x_0, O_0) \\
= 3 - 0 - 2 - B(0, 3) - 1 + \min(1, 1) = 1
\]

The modified scheduling table is shown in Fig. 7, while the modified parameters are shown in Table 5. After performing the guarantee test, the job $j_0$ is selected for execution since $a_0 = t = 0$ and the best-effort queue is empty. When the job $j_0$ completes, the scheduler gets activated again at $t = 2$ and schedules $j_1$ and later, at $t = 4$, $j_2$ to complete the execution of the scheduling table.

4.4. Offline guarantee analysis

In order to design a robust and responsive real-time system, a usual practice is to guarantee the correct behaviour of a sub-system or task offline. To guarantee some service to aperiodic activities, a certain amount of processor bandwidth is usually reserved for aperiodic tasks (or servers). Due to the limited system resources and large number of constraints, there is always a limit to the bandwidth which is considered a “safe reservation” for handling aperiodic activities. In industry, such reservations are mandatory to guarantee system safety. However, reserving a specific bandwidth for such a case still limits the service for aperiodic execution. In this work, we conjecture that, instead of reserving a limited amount of processor bandwidth, all free partition/processor bandwidth can be utilized to service aperiodic tasks. The following theorem provides the base for our conjecture:

\[
\begin{array}{ccccc}
\hline
j & r & a & f & d & C & x \\
\hline
0 & 0 & 0 & 2 & 3 & 2 & 1 \\
1 & 0 & 2 & 4 & 8 & 2 & 1 \\
2 & 0 & 4 & 9 & 10 & 2 & 1 \\
\hline
\end{array}
\]

Theorem 4.3. For a given TT schedule $S$, if there exists a reservation such that a number of non-preemptive aperiodic jobs with a defined job release pattern can be executed feasibly, the job-shifting algorithm can also execute them without a reservation.

Proof. Consider a TT scheduling table $S$ with defined activation times $a_i$ and finish times $f_i$ for each periodic job $j_i$. A reservation window $N_{(t_1,t_2)}$ with window start time $t_1$ and end time $t_2$ is defined as contiguous reserved processing time for the execution of non-preemptive aperiodic job(s). According to Lemma 4.2, the aperiodic room $R_k$ provides the maximum room for executing non-preemptive aperiodic job, where $k \in \mathbb{N}$ and $a_k \geq t_1$. When a non-preemptive aperiodic job $j_{w}$ with $C_{w} = t_{1} - t_{1}$ is released at $t_{w} \leq t_{1}$, then by definition, the condition $N_{(t_1,t_2)} \leq R_k$ always holds. If the non-preemptive aperiodic job $j_{w}$ can be feasibly executed by $N_{(t_1,t_2)}$, $R_k$ can also feasibly execute it without an offline reservation. The proof can be iteratively applied to multiple reservation windows. Hence, the theorem is proven.

In other words, the same practice of “safe reservation” can be utilized to give guarantees for aperiodic execution in job-shifting without any online reservation. However with job-shifting, the bandwidth loss due to partitioning [2] can be reduced, the aperiodic response-times can be significantly improved (further in Section 7.3) and the task jitter can be controlled (further in Section 7.7).

Being a non-preemptive algorithm, a necessary condition for aperiodic guarantee can also be checked offline in job-shifting. Independent of the release time of the aperiodic job, there must exist enough aperiodic room $R_k$ in the schedule $S_p$ to accommodate complete aperiodic job. For an aperiodic task, if there exists no aperiodic room $R_k[R_k \geq C_w]$ for any $j \in S_p$, the aperiodic job will never be able to execute while keeping feasibility of other tasks in $S_p$. This necessary condition is checked between the offline and the online phases of the job-shifting algorithm.

5. Efficiency evaluation

In this section, we present the results of the experiments performed to evaluate efficiency of the job-shifting algorithm. In Section 5.1, we provide the description of the experimental setup. While in Section 5.2, we present and briefly discuss the obtained results.

5.1. Experimental setup

In order to provide a reference point for the job execution guarantees provided by the job-shifting (JS) algorithm, we implemented a non-preemptive hierarchical background scheduler (NP–BG). The NP–BG scheduler services the aperiodic jobs during the idle time available in the partition scheduling table, i.e. a non-preemptive aperiodic job is executed only when it can be guaranteed that the job will finish within the idle time. Note that any existing algorithm (e.g. non-preemptive version of slot-shifting [6], constant bandwidth server [25], sporadic server [26], etc.) cannot be used to provide such a reference point for comparison due to varying system models (e.g. preemptive execution environment, only flat scheduling model, etc.).

For the evaluation of job-shifting algorithm, 1000 task-sets were generated. The parameter ranges for generating task-sets were selected as defined by Schorr [6]. Each generated task-set consisted of at most one processor and one partition. The partition blockings $V$ are
generated with a strict period of 10 and a relative beginning time \( b = 6 \) of each blocking \( v \in V \) until the end of the scheduling cycle \( SC \). As the number of parameters and their ranges for generating task-sets are very large, the specific periods and sizes of the partition blockings \( V \) do not lead to a biased result.

Inside the partition, \([1, 3]\) periodic tasks were generated with phases \( \phi = 0 \), WCET \( C \) in the interval \([1, 15]\) and the period \( T \) in the interval \([15, 30]\) (with implicit deadlines). The periodic task parameters were generated using UUniFast [27] algorithm to get uniform tasks distribution. The aperiodic tasks were generated with release time \( r \) in the interval \([0, SC]\), the WCET \( C \) in the interval \([5, 10]\) and the deadline \( d = DLX \times C \), where DLX is the deadline extension factor. The factor DLX defines the tightness of the deadline compared to \( C \), i.e. \( DLX = (d - r)/C \). The generation process of the aperiodic tasks is stopped when the aperiodic utilization reaches the target utilization.

To distinguish the effects of varying urgencies, task-sets were generated for each different DLX factor in the list \([4, 8, 12]\) and the aperiodic tasks utilization in the list \([5\%, 10\%, 15\%, 20\%]\). The guarantee ratios (i.e. the ratio of the number of guaranteed aperiodic jobs to the total number of aperiodic jobs) of the algorithms are also affected by the partition supply and the total periodic tasks utilization. Therefore, all the task-sets were generated for two different partition supplies in the set \([70\%, 50\%]\) and two different periodic task demands in the set \([25\%, 35\%]\).

As mentioned in Section 3, the job-shifting (JS) algorithm requires a cyclic executive schedule for the partition. Therefore during the preprocessing stage, the partition schedule \( S_p \) is generated using the EDF scheduler. The task-sets which resulted in \( SC \) lengths outside of the interval \([500, 5000]\) were rejected since the real-world tables are smaller [6]. Moreover, the task-sets were also filtered using the offline guarantee analysis method mentioned in Section 4.4.

5.2. Results and discussion

The Figs. 8 and 9 show the average guarantee ratios for the generated task-sets (i.e. each point in the figure represents the average guarantee ratio of 1000 task-sets) for varying task utilizations. In the figures, the DLX factor is represented with different pointer types and the schedulers (JS and NP–BG) with different line types. For the non-preemptive background scheduler (NP–BG), the guarantee ratio defines the ratio of the number of aperiodic jobs which finished before their deadlines to the total number of aperiodic jobs. Note that the NP–BG scheduler provides the least average guarantee ratio which can be achieved by a simple background queue.

The Figs. 8 and 9 show that, for the background scheduling, the number of finished aperiodic jobs decreases to a great extent (e.g. from 0.7 to 0.2 for DLX = 12, aperiodic task utilization \( U_{ap} = 20\% \) and periodic task utilization \( U_p = 25\% \)) due to just 20% decrement in the partition supply. However, for the job-shifting algorithm, the number of finished aperiodic jobs did not suffer as much (e.g. from 1.0 to 0.9 for DLX = 12, aperiodic task utilization \( U_{ap} = 20\% \) and periodic task utilization \( U_p = 25\% \)). When the periodic task demand is increased from 25\% (in Fig. 8) to 35\% (in Fig. 9), the guarantee ratios of all the schedulers are decreased. However, the JS scheduler provides way better guarantee ratio than the NP–BG scheduler (e.g. \( \phi = 0.1 \) for NP–BG vs \( \phi = 0.5 \) for JS for \( U_{ap} = 20\%, U_p = 35\% \) and partition supply = 50\%). Notice that for the extreme case of \( U_{ap} = 20\%, U_p = 35\% \) and partition supply = 50\% in Fig. 9b, a guarantee ratio of lower than 1 is inevitable as the partition supply is less than partition demand.

In summary, the average guarantee ratio of the job-shifting algorithm is quite large compared to the background processing. Therefore, we conclude that the job-shifting algorithm is more suitable compared to the background processing for online aperiodic job admission in hierarchic non-preemptive systems.

6. Overheads evaluation

In this section, we present the results of the experiments performed to evaluate overheads incurred by the job-shifting algorithm. In Section 6.1, we provide the description of the experimental setup. While in Section 6.2, we present and briefly discuss the obtained results.

6.1. Experimental setup

In order to evaluate the job-shifting overheads, we implemented the job-shifting algorithm in the DREAMS project. Section 6.1.1 provides the description of the hardware and software platforms used in the DREAMS project. Section 6.1.2 briefly discusses the safety critical demonstrator application. Section 6.1.3 describes how the job-shifting algorithm is integrated in the DREAMS tool-chain and finally, Section 6.1.4 defines how the safety critical application is deployed for the evaluation of the job-shifting algorithm.

6.1.1. Hardware & software platforms

The DREAMS Harmonized Platform (DHP) is a heterogeneous multi-core platform developed on top of the Xilinx Zynq ZC-706 FPGA platform. The Zynq platform provides a dual-core ARM Cortex-A9 processor running at 400Mhz. The DHP design extends the ARM processor with 3 MicroBlaze cores. The different cores are interconnected through a
A dedicated network designed to provide support for time-critical communication.

In the context of the DREAMS project, the XtratuM hypervisor [28] was ported to run on top of the ARM processor, NXP QorIQ T4240 and Intel x86. Moreover, a software abstraction layer is added between the XtratuM and the user applications to provide services like local and global resource management with support for online system reconfiguration [29]. To implement these services three special system partitions were developed: the monitor (MON), the local resource manager (LRM) and the global resource manager (GRM). Additionally, the user partitions (on which the applications are deployed) implement a local resource scheduler (LRS) which provides the applications with a cyclic-executive intra-partition scheduler (CEIPS [16]) and an interface to the developed services.

6.1.2. Safety critical demonstrator

In the context of DREAMS, a demonstrator of a safety critical system was developed by Thales to assess the technologies developed within the project. The demonstrator can be seen as a distributed system composed of three platforms (2 NXP QorIQ T4240 and a DHP) interconnected via a TTEthernet switch [29].

To implement these services three special system partitions were developed: the monitor (MON), the local resource manager (LRM) and the global resource manager (GRM). Additionally, the user partitions (on which the applications are deployed) implement a local resource scheduler (LRS) which provides the applications with a cyclic-executive intra-partition scheduler (CEIPS [16]) and an interface to the developed services.

6.1.3. DREAMS tool-chain & job-Shifting integration

In order to ease the process of artifact generation and support product updates, the DREAMS project provides a tool-chain [30] for industrial mixed-criticality applications. A detailed description of the tool-chain can be found in [30]. Here, we only provide the description of the relevant tools (see Fig. 11).

The tool AutoFOCUS 3 [31] (‘af3’ in Fig. 11) is an Eclipse-based modeling tool, which is used to define the models for the application, platform, partitioning and application deployment in terms of DREAMS meta-models [32]. Once the models are created, text file generators can be used to generate input for a specific tool in the tool-chain. After the invocation of the tool, the output file generated by the tool can be imported back in AutoFOCUS 3 in order to update the model. When all the required tools are finished updating the models, the platform configuration file(s) can be generated.

In order to generate the input required to execute the job-shifting algorithm inside partition p, the web-based tool Xoncrete [33] is used to generate the inter- and intra-partition schedules for XtratuM [28] hypervisor. Afterwards, a command-line tool called MCOSF (abbreviation of ‘Mixed-Criticality Offline Scheduling Framework’) is used, which calculates the flexibility coefficients x for each job of partition p until the end of scheduling cycle SC (defined in Section 3). The generated information is then forwarded to the local resource scheduler (LRS [34]) of partition p via Partition Configuration File (PCF) (see Fig. 11).

In this work, the partition LRS is modified to implement the job-shifting algorithm with dense time-base [12] (see Section 3) on top of

---

**Table 6**

<table>
<thead>
<tr>
<th>Application</th>
<th>Periodic tasks</th>
<th>Aperiodic tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>APP1</td>
<td>6</td>
<td>0.021</td>
</tr>
<tr>
<td>APP2</td>
<td>10</td>
<td>0.545</td>
</tr>
<tr>
<td>APP3</td>
<td>5</td>
<td>0.055</td>
</tr>
<tr>
<td>APP4</td>
<td>1</td>
<td>0.015</td>
</tr>
<tr>
<td>Total</td>
<td>22</td>
<td>0.636</td>
</tr>
</tbody>
</table>

---

*With the exception of the communication from APP3 to APP4, to evaluate overheads for partitions without any aperiodic task.

---

4 https://eclipse.org/.
cyclic-executive intra-partition scheduler (CEIPS [16]). As CEIPS does not make use of a ready-queue, a simple job dispatch table for each partition slot [28] is specified in the PCF. To detect the aperiodic job activation, the sources of the aperiodic task activation are also specified in the PCF (i.e. virtual network ports for message activated tasks, and interrupt sources for tasks activated due to external inputs). In order to avoid the implementation of dynamic memory management, the memory required for accommodating newly released aperiodic job(s) is statically reserved. Moreover, similar to the heuristic proposed by Schorr [6], at most one aperiodic job is guaranteed per LRS activation in order to bound the scheduler overheads.

Besides the constraints mentioned in Section 3, the scheduler CEIPS puts forward another constraint: A job started in a partition slot \( s_p \) [28] must finish before the end of \( s_p \). In other words, a task cannot be paused at the partition boundary. To accommodate this constraint in the job-shifting algorithm, each partition blocking \( v \in V \) (i.e. the opposite of partition slots in XtratuM [28]) can be considered a job for the guarantee test/procedure (with \( a = b, C = m - b, f = m \) and \( x = 0 \)). The advantage of such an assumption enables the use of flat scheduling model for the implementation in the LRS, in turn reducing the run-time complexity of the job-shifting algorithm (discussed in Section 7.3).

6.1.4. System deployment

Even though the DREAMS Harmonized Platform (DHP, see Section 6.1.1) provides multiple cores, only one of the ARM cores running the XtratuM hypervisor was used for the overhead evaluation. This approach enables evaluation with large enough core utilization, further exhibiting benefits of the job-shifting algorithm over reservation based or background approaches. Moreover, other DREAMS services (e.g. global resource management) were disabled as their focus is orthogonal to the overheads evaluation of this study.

For the overheads evaluation of the job-shifting algorithm, the four demonstrator applications mentioned in Section 6.1.2 were each deployed in their own user partition, while the resource management applications, i.e. the LRMs and MONs (see Section 6.1.1), were deployed in system partitions on a single ARM core of the DHP as shown in Fig. 12. Moreover, the external inputs to activate aperiodic tasks were triggered with a defined probability to simulate event occurrence. This strategy enables more frequent aperiodic activations compared to the generated job-shifting schedule (i.e. inter-partition schedule) resulting in a scheduling cycle length SC of 40 s. To provide a rough idea of the inter-party schedule, the initial part of the schedule is shown in Fig. 13, while the range of flexibility coefficients and the activation probabilities for tasks activated by external events are provided in Table 7. In the table, the activation probability of 1 means that a job of the aperiodic task is released every second.

6.2. Results and discussion

Fig. 14 shows the bar plots of the measured maximum overheads for each partition utilizing job-shifting scheduler within 1000 scheduling cycles SC (See Section 3). Fig. 14a shows the maximum overheads for the case where the schedule is implemented as a static C array, while Fig. 14b shows the same for the case where the schedule is implemented as a double linked list. As mentioned earlier, the dynamic memory management scheme is not implemented, due to which the size of the double linked list is statically defined but still permits O(1) add job operation for the guarantee procedure. In order to exhibit the difference between the two implementations, the job insert operation (the first step in guarantee procedure) is shown separately from the guarantee procedure. The error bars in Fig. 14 represent the 98% confidence interval for the cumulative overheads. The legends in Fig. 14 represent the overhead distribution, where ‘job insert operation’ overheads represent the time elapsed in inserting a newly released aperiodic job in the job dispatch table and the ‘Misc Overheads’ represents the rest of the scheduling overheads (e.g. next job selection, looping, updating pointers). It is important to mention here that the graph in Fig. 14 includes the logging overheads and excludes the overheads due to the detection of aperiodic task arrival, since these overheads are dependent on the aperiodic activation source, the hyper-calls implementation, and the platform architecture. For the aperiodic tasks activated by the reception of a message on a virtual network port, the overheads were measured to be approximately 13 µs per aperiodic task. Whereas the detection of the

### Table 7

<table>
<thead>
<tr>
<th>Application</th>
<th>Flexibility (s)</th>
<th>Activation probability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min (ms)</td>
<td>max (ms)</td>
</tr>
<tr>
<td>APP1</td>
<td>13</td>
<td>30</td>
</tr>
<tr>
<td>APP2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>APP3</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>APP4</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
APERIODIC TASKS-triggered by the external events required approximately 2 µs per aperiodic task.

Fig. 14 shows that the overheads incurred by the job-shifting algorithm strongly depend on the number of tasks inside a partition. Moreover, the overheads of the guarantee test and the guarantee procedure depend on the number of periodic tasks, the number of aperiodic tasks and the flexibility of guaranteed tasks inside a partition. Note for the double linked list implementation of the schedule (see Fig. 14b), the job insert operation took merely 3 to 4 µs. Although with this implementation the rest of the overheads are slightly increased, the cumulative overhead is decreased.

It was also observed that, on average, 22.6 aperiodic jobs were released per second during the execution of 1000 scheduling cycles SC. Furthermore, none of the released aperiodic jobs missed its deadline and all the aperiodic jobs were guaranteed for the first invocation of Eq. 4 due to relatively smaller utilizations of periodic tasks and good enough aperiodic Room R.

In 2015, Schorr [6] measured the overheads of the slot-shifting algorithm on a cycle accurate MPARM simulator running at 200 MHz. He mentioned that, for his synthetic task-set, the overheads for slot-shifting acceptance test ranged from 8.695 µs to 23.7 µs, while the overheads for the guarantee procedure ranged from 9.82 µs to 42.99 µs (excluding overheads due to job insert operation, see experiment 3 in [6]). For our safety critical demonstrator application, Fig. 14 shows that the overheads of job-shifting algorithm are comparatively smaller (i.e. 3.825 µs to 9.17 µs for guarantee test and 7.71 µs to 12.3775 µs for guarantee procedure). The smaller overheads for job-shifting on the Zynq ZC706 platform were partly expected as the operating frequency of the platform is twice compared to the MPARM. However, the overheads with job-shifting algorithm are smaller than half of the overheads with slot-shifting, which manifests that the overheads incurred by the job-shifting algorithm are quite comparable to preemptive algorithms like slot-shifting [4,6]. Note that this comparison does not indicate superiority of job-shifting algorithm over slot-shifting algorithm, due to varying system models and test scenarios.

7. Discussion

This section provides a detailed description of which factors are considered by job-shifting algorithm, which factors affect the overheads incurred by job-shifting algorithm and how this algorithm can be optimized for a system under consideration.

7.1. Mixed-critical tasks

In the industrial mixed-criticality task model defined by the standards IEC61508 [20], DO-178C [21] and ISO26262 [22], the task criticality is used to refer to the level of assurance applied to the development of software application and the different criticality tasks are temporally segregated by allocating them to different partitions [1]. For such industrial standards, the tasks only define single WCET C, the higher criticality level of a task does not mean greater importance and, therefore, does not warrant rejection of lower criticality tasks. All these facts lead to a conclusion that the task criticality designation, as per industrial standards, cannot be exploited by the scheduler. Note that the schedule can only exploit the criticality designation when critical and non-critical tasks co-exist in a single partition. In safety critical systems, the non-critical tasks are usually complex tasks (for instance, multimedia, java virtual machine [35], etc.), which handle large amount of data, consume large number of resources and are often designed carelessly. When such non-critical tasks execute in a partition alongside critical tasks, (i) the non-critical tasks need to be certified at the highest criticality level of the tasks inside that partition [1] and (ii) the critical tasks become vulnerable to attacks and, perhaps, may compromise system operation.

It is important to note in Section 4 that neither flat nor hierarchical scheduling models take the task criticality Y into account. The reason for such a deliberate elimination is the use of the industrial mixed-criticality model defined by the standards IEC61508 [20], DO-178C [21] and ISO26262 [22]. As discussed earlier, the task criticality designation Y can only be exploited by the scheduler when a partition contains both critical and non-critical tasks, which is discouraged by the standards [1] in order to provide strong spatial and temporal isolation between tasks from different criticalities.

In cases where critical and non-critical tasks must execute inside the same partition, the non-critical tasks can be dropped in favor of servicing critical aperiodic tasks [1]. To handle such a situation with job-shifting algorithm, a flag f to indicate task criticality (i.e. either critical or non-critical) needs to be added to each job in the partition scheduling table Sp. In order to drop non-critical tasks in favor of critical tasks, the guarantee test can be implemented based on one of the two heuristics: optimistic or pessimistic guarantee test heuristics.

To implement optimistic guarantee test heuristic, at first the guarantee test (i.e. R \( \geq C_{\text{up}} \)) is performed without considering the job criticality flag J. If the test fails and the next job \( i | n \leq i < l \) is a non-critical job, the guarantee test is performed after merging the flexibility of jobs \( k \) and \( k-1 \) (by defining flexibility \( x_{i-1} = \max(x_{i-1} + C_i + x_i - O_i, d_i - a_i - O_i) \) and ignoring job J). If the test passes, the job i is dropped and the guarantee procedure is performed.

To implement pessimistic guarantee test heuristic, in the first stage the guarantee test assumes all non-critical jobs to be dropped (by merging the flexibilities x similar to the optimistic guarantee test heuristic), but only drops the non-critical job(x) when the test passes. Once the test passes, it can be checked (when required) if the non-
critical job can still be executed.

The problem with optimistic and pessimistic heuristics is large (implementation and run-time) complexity and memory overheads. Therefore, it is discouraged to use such optimizations in safety critical systems and, thus, are not considered an integral part of the job-shifting algorithm.

7.2. Hypervisors and clocks

There exists a multitude of real-time hypervisors in the market today. Based on the type of interface they provide to the partition, the online complexity of the job-shifting algorithm can be modified. Some hypervisors, e.g. XtratuM [28], provide a partition local clock which ticks only when the partition is active. In such hypervisors, the equations for the online phase of the flat scheduling model can be used in a hierarchical design by defining $a_i$, $r_i$, and $d_i$ in terms of the partition local clock. For such an implementation, the adjust operator $A(u)$ can be completely eliminated and the blocking operator $B(p, q)$ is only required for the last step of guarantee procedure (where the flexibility coefficient is calculated). However, there exist hypervisors which do not provide a partition local clock, e.g. PikeOS,\(^6\) and therefore need to implement both operators $A(u)$ and $B(p, q)$.

7.3. Feasibility & complexity

When the scheduler needs to adjust a number of aperiodic jobs, the problem of guaranteeing all the aperiodic jobs becomes a variant of bin-packing problem. The decision version of the bin-packing problem is known to be NP-complete [36]. Moreover, a non-clairvoyant online problem of guaranteeing all the aperiodic jobs becomes a variant of bin-packing problem. Liu and Layland [37] recommend worst-case heuristics for all the jobs can be calculated offline. Depending on the required performance metric, the guarantee test can utilize any bin-packing heuristic, e.g. first-fit, best-fit or worst-fit. The first-fit heuristic can be used when least aperiodic response-time and online overheads are required. In such a heuristic, the guarantee test starts from job $j_i$ and inserts the newly released aperiodic job before the first job with enough aperiodic room $R_i$. On the contrary, when larger online overheads can be tolerated but better distribution of aperiodic jobs is required, the best- or worst-fit heuristics can be utilized. Moreover, the order in which multiple aperiodic jobs are guaranteed also impacts the overall feasibility of jobs. Lupu et al. [37] recommend worst-fit decreasing utilization heuristic in order to maximize the number of guaranteed tasks.

For the flat scheduling model, the guarantee test and the guarantee procedure have a complexity of $O(m)$, where $m$ is the number of jobs activated during the interval $[t, d_{wp})$, which is not different from the slot-shifting algorithm [4]. Depending on the required performance metric, the guarantee test can utilize any bin-packing heuristic, e.g. first-fit, best-fit or worst-fit. The first-fit heuristic can be used when least aperiodic response-time and online overheads are required. In such a heuristic, the guarantee test starts from job $j_i$ and inserts the newly released aperiodic job before the first job with enough aperiodic room $R_i$. On the contrary, when larger online overheads can be tolerated but better distribution of aperiodic jobs is required, the best- or worst-fit heuristics can be utilized. Moreover, the order in which multiple aperiodic jobs are guaranteed also impacts the overall feasibility of jobs. Lupu et al. [37] recommend worst-fit decreasing utilization heuristic in order to maximize the number of guaranteed tasks.

For the flat scheduling model, the guarantee test and the guarantee procedure have a complexity of $O(m)$, where $m$ is the number of jobs activated during the interval $[t, d_{wp})$ (or $l - n$). For the hierarchical scheduling model, we introduced two operators $A(u)$ and $B(p, q)$. A naive implementation of these operators has a complexity of $O(r)$, where $r$ is the cardinality of the blocking set $V$. When partition local clock is available, the complexity of the guarantee test does not change. However, the complexity of the guarantee procedure changes to $O(mr)$. On the contrary, when the partition local clock is not available, the complexity of the guarantee test becomes $O(mr^2)$, while the complexity of the guarantee procedure changes to $O(mr^2)$.

7.4. Optimisations

In Section 3, it was assumed that all the tasks execute for their complete worst-case execution time $C$. However, this assumption seldom holds during system operation. A task executing for more than $C$ can lead to the violation of the temporal isolation between tasks of the same application. In the worst case, such violations may never lead to a processor yield for other tasks to execute. To protect from such a situation, a hardware timer can be programmed to generate an overrun interrupt which terminates the misbehaving job and returns the control to the scheduler. When the job executes less than $C$ time, the overrun interrupt can be terminated (or rescheduled for the next job). In such a situation, the free processing time can be utilized by using a simple approach. At the observed job finish time $t$, if for the next job $a_{n+1} > r_n$, the flexibility coefficient $x_n$ can be updated to $x_n + (a_n - \max(t, r_n))$ and the job can be preponed to activate at max ($t, r_n$). If for the next job $a_{n+1} = r_n$, the free resources can be accounted for by the aperiodic room $R_n$ and therefore, require no further action.

When a large number of aperiodic jobs are released before the scheduler activation, performing guarantee procedure for all of them may lead to large scheduler overheads. In the worst case, the reserved processing time to execute the next job might be reduced, leading to a job incompletion. To avoid such a scenario, Schorr [6] proposed a heuristic where at most one aperiodic job is guaranteed for each activation of the scheduler.

When the guarantee test passes or when the schedule repeats after SC, an insert operation is required on the scheduling table $Sp$. Therefore, the scheduling table is recommended to be stored as a double linked list, due to its $O(1)$ insert operation complexity. However, the blocking set $V$ can be stored as a contiguous vector as random access may be required. All the hypervisors supporting cyclic scheduling policy for partition scheduling store the blocking set $V$ or the partition availability set $V'$ but may not expose it to the partition. A simple API can be added to open-source hypervisor, e.g. XtratuM [28], to expose this information. In order to save the overheads related to the hypercalls, the partition blocking set $V$ can also be stored in partition local memory.

7.5. Intra-partition scheduling

In the job-shifting algorithm, the degree of flexibility of a job is defined by the flexibility coefficient. If the flexibility coefficient for all the jobs is zero, the job-shifting algorithm reduces to the background processing approach. Therefore, it can be said that the flexibility coefficient is the direct measure of the adaptability of a schedule. Depending on the scheduler used prior to the offline phase of job-shifting, the flexibility coefficient can vary significantly. The EDF approach results in the largest flexibility coefficients, while the latest-deadline-first (LDF) results in the least.

In Section 3, the input for the offline phase of job-shifting was defined to be a feasible TT partition schedule $Sp$. However, this restriction can be relaxed when (i) the online scheduling strategy is known and (ii) it can be made sure that the estimated order of execution of jobs will not be violated online. For such an online scheduler, the flexibility coefficient for all the jobs can be calculated offline. Nonetheless, the online scheduling of non-preemptive periodic tasks is an NP-Hard problem [11] even with harmonic periods [38] or with arbitrary period ratios [39]. Due to the complexity of the problem, a feasible TT partition schedule $Sp$ seems a better choice as an offline exhaustive search can be done (perhaps with large search time) with minor efforts to generate such a schedule.

7.6. Memory requirement

As the amount of memory required by an algorithm is strongly dependent on the processor clock frequency, the word size (e.g. 32-bit) and the required resolution of timely parameters (e.g. $a_i$, $d_i$), we will measure the memory overheads in terms of variables, instead of bytes. Moreover, for the two implementations, i.e. cyclic-executive [16] intra-partition scheduler with bandwidth reservation for aperiodic tasks (CEIPS-BR) and CEIPS with job-shifting (CEIPS-JS), only the structures

---

\(^6\) https://www.sysgo.com/products/pikeos-hypervisor/.
which vary in size will be discussed.

7.6.1. Implementation: CEIPS–BR

For the simple case of CEIPS–BR, there exist only one intra-partition scheduling table per partition, containing jobs for periodic as well as aperiodic (as periodic server/reservation) tasks. The scheduling table is defined as a static array of entries, each defining a job using the tuple $\langle a, r \rangle$ (where $r$ represents the pointer to the task code to which the job belongs to). Note that there is no need to store the partition blockings $V$ for this implementation.

7.6.2. Implementation: CEIPS–JS

For a naive implementation of CEIPS–JS, two intra-partition scheduling tables per partition are required, one for reloading the new table and one for job-shifting. Note that the first scheduling table used for reloading in CEIPS–JS only contains jobs for periodic tasks. The scheduling tables are defined either as a static array or as a static double linked list of entries, each defining a job using the tuple $\langle a, C, d, x, \tau \rangle$. It is important to mention here that when the schedule is implemented as a double linked list, two pointers for left and right elements of the list are also needed. When the hierarchic scheduling model is used, the job entry in the scheduling tables also includes the job finishing time $f$. Moreover, for the hierarchic scheduling model the partition blockings $V$ are stored as a static array of entries each defined by the tuple $\langle d, m \rangle$. Other than the two scheduling tables (for a naive implementation), a third table is required to store the information about the aperiodic tasks. The entries in this table are stored as the tuple $\langle D, C, S, \tau \rangle$ (where $S$ represents the source of aperiodic task activation, e.g. interrupt flag).

In summary, even for the best-case, CEIPS–JS requires 3 more variables per periodic job (with two copies of the tables for a naive implementation) and 2 more variables per aperiodic task compared to CEIPS–BR. However, note that the scheduling table in CEIPS–BR stores entries for all aperiodic jobs, which is not the case with CEIPS–JS.

7.7. Jitter control

In control systems, for instance, the task jitter has a strong impact on the overall performance of the system. Moreover, a task executing too early or too late (within its execution window, i.e. $d - r$) can lead to large variations in end-to-end latencies for large task chains. Although the worst-case jitter for a task can be estimated offline for non-preemptive priority-based scheduling approaches [9], for instance npRM, the task jitter cannot be controlled without modifying the priority of a task. Moreover, modification of the priority of a single task may affect all tasks. On the contrary, offline TT scheduling approaches can be used to have a fine control over the task jitter.

As mentioned in the previous section, the flexibility coefficient $x$ for a job in job-shifting algorithm provides a direct measure of the execution freedom the job has in the offline schedule. By limiting the flexibility coefficients of all the jobs of a task, the jitter can be controlled. In other words, the following condition must be satisfied either by construction of the partition schedule $S_p$ or by decreasing the job flexibility coefficients $x$.

$$\forall j, t_j \in T; a_j - r_j + x_j \leq e_j$$

(8)

where $e_j$ is the desired activation jitter of the periodic task $r_j$. Moreover, the activation jitter of an aperiodic task can be controlled online during the guarantee test by verifying the condition: $x_j - r_j + x_j \leq e_j$ (see Eqs. (4) and (7)). Note that limiting jitter for a task in job-shifting might potentially lead to reduced jitter of jobs executed before the task under consideration.

7.8. Free resources management

After generating the offline scheduling table $S$ for the system, there might exist scheduling holes (i.e. time duration which is not used by any partition). Note that in our system model (defined in Section 3), the hypervisor is not aware of the released aperiodic jobs and the partition is not aware of the scheduling holes as they reside outside of the partition scheduling table $S_p$. If an online scheme for management of free resources is required, either hypervisor needs to be aware of the aperiodic tasks or the partition needs to be aware of the scheduling holes, so that the partition slots can be stretched or shrunk online (similar to flattening method by Lackorzynski et al. [2]). However in this case, it is very difficult to provide offline guarantees and the cyclic execution strategy (recommended by certification authorities [1]) needs to be modified. Therefore, it is recommended that these scheduling holes are managed offline.

Depending on the hypervisor interface and the application specifications, there exist a number of offline schemes for the management of free resources. An easier and viable scheme for offline management is to enlarge the existing partition slots (where possible) to consume the scheduling holes or to make new partition slots (where enlargement is not possible) for the preferred partition. This method can help avoid modifications to the cyclic execution scheduler and provide flexibility in selecting which partition needs to have more free time. If the hypervisor switches to an idle partition during the scheduling holes, another solution is to use the idle partition to service aperiodic jobs using job-shifting algorithm as defined in Section 4. However, this solution might require hypervisor modifications and, therefore, it is not recommended. On the contrary, if the hypervisor stays in the kernel/privileged mode during the scheduling holes, the aperiodic tasks cannot execute in the context of the hypervisor and, hence, such a solution cannot be implemented.

8. Conclusion

Due to the increasing demand of functionality, modern real-time applications are required to incorporate ever increasing number of features on a single platform. To reduce the verification and validation costs for such a system, hypervisors are utilized to temporally and spatially partition the resources among the applications. However, partitioning is prone to bandwidth loss [2], due to which the effective processor utilization can be significantly reduced when bandwidth reservation techniques are used to service aperiodic tasks. Moreover, the state-of-the-art approaches to service aperiodic activities incur significant scheduling overheads and lack support for partitioning, industrial mixed-criticality task model and non-preemptive task execution.

In this paper, we presented the job-shifting algorithm for online admission of non-preemptive aperiodic tasks in hierarchical systems. We demonstrated that our algorithm provides offline guarantees similar to the bandwidth reservation technique, and therefore, no modification is required in the certification process. We also provided a necessary test for the admission of non-preemptive aperiodic tasks. We evaluated the efficiency of our algorithm on synthetic but practical task-sets and demonstrated that our approach efficiently utilizes the available resources. Moreover, we implemented the job-shifting algorithm on an ARM based platform and evaluated the incurred scheduling overheads. The experiments manifested that the overheads incurred by our scheduler are very small and practical for the safety critical applications.

In future, we plan to extend the job-shifting algorithm for parallel execution of mixed-critical tasks on multi-core platforms, provide support for mode-changes and evaluate the impact of job-shifting algorithm on the performance of the multi-core safety critical application.

Acknowledgment

This research was partly supported by the project DREAMS (Distributed REal-time Architecture for Mixed criticality Systems) from the European Union’s Seventh Framework Programme FP7/2007–2013 under grant agreement no. 610640.
References


Ali Syed is a researcher and a PhD student at the chair of Real-Time Systems in Technische Universität Kaiserslautern (TUK), Germany. He acquired his Masters degree in Electrical and Computer Engineering from TUK. His research interests include offline scheduling, model-based system design and adaptive real-time systems. Syed is actively involved in the European FP7 DREAMS project as the scheduling task leader and has authored several papers in international conferences and workshops. He is also responsible for the real-time systems lab and has supervised numerous bachelor and master theses at the chair of real-time systems.

Daniel Gracia Pérez is a Research Engineer at THALES (France) with a PhD on Computer Architecture from Paris XI University and engineering degrees from the Kungliga Tekniska Högskolan (Sweden) and the Universidad Politécnica de Catalunya (Spain). Previous professional experiences include the CIRA (France), Philips (France) and Ericsson (Sweden/France). He participated in various French and European research projects, and authored numerous articles at international conferences and journals. His current research includes the development and application of new multicore architectures for safety-critical systems. His research interests include computer architecture, network-on-chip design, simulation, genetic algo-

Gerhard Fohler is the leader of the chair of Real-Time Systems in Technische Universität Kaiserslautern, Germany. He received his PhD from Vienna University of Technology for the work on the MARS system. He then worked at the University of Massachusetts (Amherst) as a post-doctoral researcher within the SPRING project. He has been a researcher at Humboldt University Berlin (1996–97), investigating issues of adaptive reliability. He is currently the chairman of the Technical Committee on Real-Time Systems of EUROMICRO. He worked at Mälardalen University, Sweden (1997–2006), as the leader of the predictably flexible Real-Time Systems group.