Resource Sharing in a Hybrid Partitioned/Global Scheduling Framework for Multiprocessors

Sara Afshar*, Moris Behnam*, Reinder J. Bril*,‡, Thomas Nolte*
*Mälardalen University, Västerås, Sweden
‡Technische Universiteit Eindhoven, Eindhoven, Netherlands
Email: {sara.afshar, moris.behnam, reinder.j.bril, thomas.nolte}@mdh.se, r.j.bril@tue.nl

***Abstract*—For resource-constrained embedded real-time systems, resource-efficient approaches are very important. Such an approach is presented in this paper, targeting systems where a critical application is partitioned on a multi-core platform and the remaining capacity on each core is provided to a non-critical application using resource reservation techniques. To exploit the potential parallelism of the non-critical application, global scheduling is used for its constituent tasks. Previously, we enabled *intra*-application resource sharing for such a framework, i.e. each application has its own dedicated set of resources. In this paper, we enable *inter*-application resource sharing, in particular between the critical application and the non-critical application. This effectively enables resource sharing in a hybrid partitioned/global scheduling framework on multiprocessors. For resource sharing, we use a spin-based synchronization protocol. We derive blocking bounds and extend existing schedulability analysis for such a system.**

I. Introduction

The trend from the traditional single-core[1] processors to multi-core processors for embedded systems, demands for a proper scheduling framework for multiprocessors. For embedded real-time systems, which are resource constrained, such a framework must support resource-efficient approaches. From an industrial point of view, co-existence of multiple independently developed real-time applications on a shared multi-core platform is an effective and a resource-efficient solution, since it allows re-usability of applications and decreases system power consumption and costs. Whereas each application initially had the entire platform at its disposal, a move towards a shared multi-core platform may result in *inter*-application resource sharing, however, e.g. operating system primitives, buffers, and memory mapped I/O. Moreover, these applications may have different criticality levels. In this paper we consider two criticality levels , i.e. *critical* and *non-critical*, and we present an approach enabling resource sharing between critical and non-critical applications on a multi-core platform.

Traditionally, two scheduling approaches exist for multiprocessor systems called *partitioned* and *global* scheduling. Under partitioned scheduling, tasks are statically assigned to processors at design time and will only execute on those processors during run-time. Under global scheduling, tasks are selected from a system wide unique global queue at run time and are scheduled on any available processor. Although global scheduling better utilizes the processors' capacity compared to partitioned scheduling, it introduces more overhead due to potential migrations of tasks among processors.

In practice, such as the automotive industry [1], critical applications are partitioned on multi-core platforms. Resource-efficient solutions for embedded systems suggest to further utilize the remaining capacity on each core. In this paper, we target such a resource-efficient platform where a critical application is partitioned and the remaining capacity on each core is made available to a non-critical application through resource-reservation techniques. Resource reservation techniques (servers) are used to guarantee temporal isolation between critical and non-critical applications which can bound the interference of non-critical application to the critical application. To exploit the potential parallelism of the non-critical application on the multi-core platform, global scheduling is offered to schedule its constituent tasks.

A specific instantiation of such a framework has been studied in [29] assuming tasks share no resources, except the CPU. In [3], we enabled *intra*-application resource sharing assuming each application has a dedicated set of resources. In this paper, we enable *inter*-application resource sharing. This gives rise to three challenges. Firstly, resource-reservation techniques provide temporal isolation for a core but not for other shared resources. Secondly, existing synchronization protocols cannot be used without modification due to hybrid partitioned/global scheduling structure. Thirdly, blocking terms have to be bounded, and derived bounds have to be incorporated in the existing response time analysis introduced in [29]. We briefly consider the first two challenges in more detail below.

Enabling inter-application resource sharing endangers the predictability of the critical application, because a task of the non-critical application may use the resource for a longer time than anticipated. To address this problem, we assume that our platform supports three different types of shared resources: (*i*) resources with guaranteed bounds on access times, such as operating system primitives, (*ii*) abortable resources, where access can be aborted upon an overrun of anticipated access times, making the resources available to other tasks [27], [19], and (*iii*) resources where a roll-back mechanism can be used upon an overrun, similar to [5]. The latter two types of resources require a monitoring mechanism, similar to resource

This work is supported by the Swedish Foundation for Strategic Research via the research program PRESS, the Swedish Knowledge Foundation and ARTEMIS Joint Undertaking project EMC2 (grant agreement 621429).

[1]In this text we will use core and processor interchangeably.

978-1-4673-7711-9/15/$31.00

reservation. To accommodate for the overhead of a roll back in the analysis, the anticipated resource-access time can be inflated with the maximum overhead of a roll-back.

The existing synchronization approaches, which have been presented for either partitioned or globally scheduled systems, cannot be used without modifications for our hybrid partitioned/global framework. In this work, we provide a resource-sharing protocol based on FIFO-queues and a spin-based locking technique, where a task performs a busy-wait loop (called also spin) whenever a request on a global resource cannot be immediately satisfied. An important quality of spin-based protocols is that on every processor only one task at the time can request and access a global resource. Preserving that quality guarantees several interesting properties such as: (*i*) the size of the global FIFO queues are bound to the number of processors in the system, (*ii*) a task may experience blocking from at most one resource access of any lower priority task on the same core, and (*iii*) one stack can be used for the tasks statically allocated to a core.

Contributions: In this paper, we enable resource sharing for a hybrid partitioned/global scheduling framework. We modify existing spin-based protocols such that interesting properties of spin-based approaches are maintained for our framework. We bound the blocking delays incurred to the tasks in the system and incorporate these bounds in the existing schedulability analysis proposed for this hybrid system.

The rest of this paper is structured as follows: Section II summarizes the existing related works in this context. Section III defines the model of our system and provides the resource sharing rules. Sections IV presents an overview of the existing analysis and spin-based approach. Sections V and VI present the blocking bounds incurred to partitioned and globally scheduled tasks, respectively. Sections VII and VIII present the new response time analysis based on the resource sharing parameters and the schedulability test steps for this framework, respectively. Finally, Section IX concludes.

II. Related Works

Hierarchical scheduling in combination with resource reservation techniques for multiprocessor platforms has been studied [14], [26], [20], [29] where it is assumed that tasks are independent and they do not share any resources other than the CPU. A significant amount of work has been presented in the context of multiprocessor resource sharing. In the following, we briefly present the most related synchronization protocols for multiprocessor systems.

The Distributed Priority Ceiling Protocol (DPCP) is a suspension-based synchronization protocol presented in [23] which has been developed for partitioned static priority scheduling. The Multiprocessor Priority Ceiling Protocol (MPCP) which is a variant of the Priority Ceiling Protocol (PCP) [25] for multi-core platforms was introduced for partitioned systems [24], [23]. MPCP is a suspension-based protocol. The Multiprocessor Stack Resource Policy (MSRP) which is an extension of the Stack Resource Policy (SRP) [6] for multiprocessors has been introduced in [17] for partitioned systems. MSRP is a spin-based approach.

The Flexible Multiprocessor Locking Protocol (FMLP) has been introduced in [9] under two variants for partitioned and global scheduling respectively. FMLP uses spin-based approach and suspension-based approaches for short and long resources, respectively. The partitioned FMLP was extended for fixed priority scheduling in [10]. A synchronization protocol called $O(m)$ Locking Protocol (OMLP) which is a suspension-based approach, has been proposed in [11], for both partitioned and global scheduling.

The Multiprocessor Synchronization Protocol for Open Systems (MSOS) presented in [22], is a suspension-based preemptive synchronization protocol that has been developed for compositional independently-developed real-time applications. Later in [4] MSOS was extended for priority-based applications which has shown improvement in the schedulability performance. In [12] resource sharing for cluster-based scheduling has been presented. Under cluster-based scheduling tasks are bounded to clusters of processors and are scheduled globally within each cluster. In [16], a schedulability analysis based on worst-case response times has been presented under fixed priority global scheduling for PIP and Parallel PCP (P-PCP) synchronization protocols.

All aforementioned synchronization protocols have been introduced either for partitioned or globally scheduled systems, and cannot be used under the hybrid scheduling framework presented in this paper.

III. System Model

A. Task Model

Each task τ_i is presented by $< C_i, D_i, T_i >$ and is constituted of an infinite sequence of jobs. C_i is the worst-case execution time of any job of τ_i, D_i is the relative deadline and T_i is the minimum inter arrival time between two successive jobs of task τ_i. We assume a constrained-deadline task model, i.e., $D_i \leq T_i$. The priority of a task τ_i is denoted by ρ_i, where $\rho_i > \rho_j$ if $i > j$. a_i and d_i denote the arrival time and the absolute deadline of any job instant of a task τ_i, respectively.

B. Architecture and Scheduling Strategy

Our system model consists of m identical unit capacity processors. The systems consist of two different types of tasks: (*i*) tasks which are partitioned over the platform and are referred to as non-migrating tasks, and (*ii*) tasks which are scheduled globally within a set of servers which are referred to as migrating tasks. The remaining capacity on each core (if any) is provided to migrating tasks by means of a server on that core which uses a similar technique as synchronized deferrable servers (similar to [29]). For the sake of presentation simplicity we assume that each core accommodates one server which has the highest priority on the core similar to [29]. However, the model and accompanying analysis can be generalized to the case where server may have any arbitrary priority as in [28]. This maybe helpful if some tasks related to the set of partitioned tasks have tight finalization jitter constraints due to belonging to a critical application. Thus, to remove the effects of induced jitter by the server, the server may get a priority lower than that of such tasks on the core. Moreover, the analysis can also be extended to accommodate multiple servers per core similar to [28].

In this work, since inter-application resource sharing is enabled, deferrable servers cannot be used precisely as in [29] without adjustments. We have provided refinements to the technique of these servers by Definition 6 in Section III-D and Rules 15 and 16 in Section III-E5 so that we can use them under our resource sharing approach. We use S_{P_k} to denote the server dedicated to a processor P_k. The total budget of a server S_{P_k} is denoted by C_{P_k}. For ease of presentation, a common replenishment period T_s is used for all servers similar to [29], however each server can have a different replenishment period.

Moreover, the set of migrating and non-migrating tasks in the system are denoted by $\mathcal{T}^{\mathrm{m}}$ and $\mathcal{T}^{\mathrm{nm}}$, respectively. The set of non-migrating tasks allocated to a processor P_k is denoted by $\mathcal{T}^{\mathrm{nm}}_{P_k}$. The priority of non-migrating tasks related to applications are assigned according to a fixed priority algorithm (e.g. the rate monotonic (RM) technique) on a processor. The priority of the server S_{P_k} is denoted by ρ_{S_k} and is the highest priority on the core, i.e., higher than the highest priority in the set $\mathcal{T}^{\mathrm{nm}}_{P_k}$ (see Def. 1). The reason is that, based on the resource efficient model of the scheduling framework used here, the budget of the server on a core is determined by the available slack from the non-migrating tasks on that core. Thus, the interference from the server to non-migrating tasks on the same core does not jeopardize the scheduling of those tasks. Further, by setting the priority of the server to the highest on a processor, no extra local blocking delay is imposed to the non-migrating tasks from the server on the same core. Note that, non-migrating tasks and migrating tasks belong to different applications, therefore, the priority of non-migrating tasks is irrelevant from those of migrating tasks.

The partitioning technique used for non-migrating tasks is not the focus of this work and we leave it as a future optimization step. Due to the complexity of the problem, for the first step, we assume that there exists an allocation solution.

C. Resource Sharing Parameters

Two types of resources may exist in the system: *local* and *global* resources. Local resources are those that are accessed by tasks on the same processor, only whereas global resources are accessed by tasks on more than one processor. By such definition, local resources are used by non-migrating tasks only. Further, all resources accessed by migrating tasks are global resources by definition. $\mathcal{R}^{\mathrm{L}}_{P_k}$ denotes the set of local resources that are accessed by (jobs of) tasks on a processor P_k. $\mathcal{RS}^{\mathrm{L}}_i$ and $\mathcal{RS}^{\mathrm{G}}_i$ denote the set of local and global resources accessed by jobs of a task τ_i, respectively. Moreover, the longest execution time among all requests of any job of a task τ_i on a resource R_q is denoted by $Cs_{i,q}$. Maximum number of requests for any global resource and a specific global resource R_q of a task $\tau_i \in \mathcal{T}_{P_k}$ are denoted by n^{G}_i and $n^{\mathrm{G}}_{i,q}$, respectively.

We use $\mathcal{RS}_i$, n_i and $n_{i,q}$, to denote the set of resources accessed by jobs of a task τ_i, the maximum number of resource requests of any job of a task τ_i on any resource and on a specific resource R_q, respectively. Nested access of resources is not the focus of this paper and is not considered here, however it can be supported by using group locks similar to [9].

D. General Definitions

In the following a set of definitions are presented which will be used in the rest of this paper. The definitions that are designed only for this work has been specified. We provide Def. 6 to make sure that a migrating task completes its resource access in case of server budget depletion to prevent excessive possible blocking delays as well as accelerating the release of a resource.

Def. 1. *(new) The highest priority on a processor P_k is denoted by $\rho^{\max}_{P_k}$ and since based on our system model the server on the core (if any) has the highest priority, it is denoted as follows:*
$$\rho^{\max}_{P_k} = \rho_{S_k} + 1. \tag{1}$$

Def. 2. *(new) The highest priority within migrating tasks is denoted by $\rho^{\max}_{\mathcal{T}^{\mathrm{m}}}$ and is presented as follows:*
$$\rho^{\max}_{\mathcal{T}^{\mathrm{m}}} = \max_{\forall \tau_i \in \mathcal{T}^{\mathrm{m}}} \rho_i. \tag{2}$$

Def. 3. *([6]) Ceiling-based resource-access protocols assign a resource ceiling to any local resource $R_\ell \in \mathcal{R}^{\mathrm{L}}_{P_k}$, where $ceil_{P_k}(R_\ell) = \max\{\forall \rho_i |\ \tau_i \in \mathcal{T}^{\mathrm{nm}}_{P_k} \wedge R_\ell \in \mathcal{RS}^{\mathrm{L}}_i\}$.*

Def. 4. *([2], adjusted) The maximum time for a task running on a processor P_k that needs to spin to acquire a global resource R_q, which is held on a remote processor, is referred to as spin-lock time for resource R_q. The spin-lock time of a non-migrating task on P_k for R_q is denoted by $spin_{P_k,q}$ and for a migrating task $\tau_i \in \mathcal{T}^{\mathrm{m}}$ is denoted by $spin_{i,q}$.*

Def. 5. *([2]) The maximum time for a task τ_i that needs to spin to acquire all its global resources is referred to as spin-lock time of task τ_i and is denoted by $spin_i$.*

Def. 6. *(new) The total budget of a server is comprised of two parts: (i) normal execution budget, and (ii) overrun budget[2]. We denote the normal execution budget of a server on processor P_k by $C^{\mathrm{nrm}}_{P_k}$ and the overrun budget as $C^{\mathrm{ovr}}_{P_k}$, where $C_{P_k} = C^{\mathrm{nrm}}_{P_k} + C^{\mathrm{ovr}}_{P_k}$.*

Based on Definition 6, $C^{\mathrm{ovr}}_{P_k}$ is the maximum spin-lock time for any resource of any migrating task plus the maximum worst-case critical section length for any resource of any migrating task as presented in below.
$$C^{\mathrm{ovr}}_{P_k} = \max_{\substack{\forall i,q: \tau_i \in \mathcal{T}^{\mathrm{m}} \\ \wedge R_q \in \mathcal{RS}^{\mathrm{G}}_i}} (spin_{i,q} + Cs_{i,q}). \tag{3}$$

The budget of a server on a processor P_k, i.e., C_{P_k}, is calculated based on the remaining slack on the core which will be explained later in Section VIII.

E. Scheduling and Resource Sharing Rules

In this section we present the scheduling and resource sharing rules used for the hybrid framework presented in this paper. For scheduling tasks within this framework a two level hierarchical intra-core and inter-core scheduling is used. We use a spin-based resource sharing approach similar to MSRP [17] where a task spins with the highest priority on a core when it is waiting for a resource. Similar to MSRP, we use FIFO-based queues to enqueue the requests of those tasks.

[2]The overrun budget is used in a similar way as in [18], [15] and [7].

1) Intra-Core Scheduling: We model a server S_{P_k} as a task on P_k with execution time C_{P_k} and period T_s. By means of such a view, the intra-core scheduling approach schedules non-migrating tasks along with the server (if any) dedicated on each core by using uniprocessor fixed-priority preemptive scheduling.

2) Inter-Core Scheduling: The inter-core scheduling approach schedules migrating tasks within the servers using global scheduling. We use a similar scheduling approach as in [29]. For the sake of completeness we present the rules of this scheduling in the following. Rules 1 and 2 are similar rules as in [29]. Rule 3 is a new rule and is provided due to resource sharing.

Rule 1. *Migrating tasks are scheduled among servers from a global priority-ordered ready queue using a priority-based preemptive scheduling. Migrating tasks are added to the queue after they are released or preempted.*

Rule 2. *If multiple servers belonging to an application are available (i.e. the server is not preempted and it has remaining capacity) the highest priority (ready) migrating task is scheduled in the server with the largest capacity.*

Rule 3. *After a migrating task releases a resource if the normal budget of the server has been depleted, the migrating task is preempted and re-scheduled among the available servers.*

3) Resource Sharing Among Non-Migrating Tasks: Since from a core scheduling point of view, servers behave similar to tasks, therefore, a spin-based resource sharing approach identical to MSRP [17] is used for non-migrating tasks on the core level. Next, for the sake of protocol completeness, we briefly recapitulate the resource sharing rules of such spin-based approach that is conformed for our system model.

Rule 4. *Local resources are handled by means of a uniprocessor synchronization protocol e.g. SRP or PCP.*

Rule 5. *For each global resource a FIFO-based queue is used to enqueue the tasks waiting for the related resource.*

Rule 6. *Whenever a task $\tau_i \in \mathcal{T}_{P_k}^{\mathrm{nm}}$ (i.e., a non-migrating task) requests a global resource which is held by another task on a different processor, it places its request in the associated resource queue and performs a busy wait (also called spin). The task spins with $\rho_{P_k}^{\max}$ priority level (see Definition 1).*

Rule 7. *The priority of the task is changed to its normal (original) priority as soon as it releases the global resource.*

Rule 8. *When the resource is released, the task at the head of the resource global queue (if any) resumes and locks the resource.*

4) Resource Sharing Among Migrating Tasks: Similar to non-migrating tasks, we use a spin-based resource sharing approach for migrating tasks as well. One important property of a spin-based approach is that only one pending request on a global resource can exist at any time on any processor (Lemma 12 in [2]). Due to the hybrid structure of partitioned and global scheduling of our system model, adjustments to spin-based protocol is required to guarantee this property. In order to preserve such a property, it is desirable that migrating tasks have a similar behavior as partitioned tasks when they block on a resource. Therefore, the rules in this section has been provided to fulfill this property. We will show later by Lemma 3 in Section V, how this property is maintained. As mentioned in Section III-C, by definition all resources requested by migrating tasks are global resources. Therefore Rule 4 does not apply for migrating tasks, whereas Rules 5, 7, 8 are applied also for migrating tasks. We extend Rule 6 to Rules 9, 10 and 11 for migrating tasks.

Rule 9. *Whenever a migrating task τ_i requests a global resource, the priority of the task is boosted to $\rho_{\mathcal{T}^{\mathrm{m}}}^{\max}$ (Def. 2) and if the request is not satisfied, i.e., the resource is held by another task, it places its request in the associated resource queue and spins.*

Rule 10. *A migrating task consumes the normal execution budget of a server while spinning if the server has normal execution budget left. Otherwise it will consume from the overrun budget to spin.*

Rule 11. *If a migrating task is granted access to a resource but the normal execution budget of the server is finished, the task consumes from the overrun budget of the server to execute its critical section.*

5) Server Rules: As mentioned in Section III-B, for scheduling the migrating tasks we use servers that are similar to deferrable servers. However, since resource sharing is used here, we have extended the rules of such servers so that they can be used under our system model. In this section we recapitulate the rules of such servers and present new rules (Rules 15 and 16) that are adjusted according to our resource sharing rules.

Rule 12. *In each server period, ready tasks run in the server by consuming the available (normal) server budget until the budget is depleted. If there is no workload, the (normal) server budget is preserved.*

Rule 13. *In each server replenishment period the total budget of the server on a processor P_k is replenished to C_{P_k}.*

Rule 14. *If there is no pending workload, the server is suspended.*

Rule 15. *If until the end of a replenishment period, normal execution budget is left, both the remaining execution budget as well as the overrun budget are discarded.*

Rule 16. *As soon as the task, which is consuming from the overrun budget, releases its resource the remaining overrun budget of the server is discarded.*

IV. Overview of Existing Approaches

In this section we briefly present a recap of the server-based scheduling analysis without resource sharing presented in [29] and spin-based resource sharing in sections IV-A and IV-B, respectively.

A. Response Time Analysis of Migrating Tasks

The response time analysis for server-based scheduling assuming that tasks are independent (i.e. without any resource sharing) has been studied in [29]. According to this analysis, a job's scheduling window (i.e. when the job is released until it finishes which is the response time interval of the job) is divided into two intervals called *head* and *body*. The head of a job is the interval between the arrival of the job and the first server replenishment, and the body is the rest of the interval, as shown in Figure 1 in Section VII. The worst-case response time of a task τ_i is specified by the response time of τ_i's critical instant. Following this, the head and body of the critical instant of τ_i is called the critical head and the critical body denoted by H_i^{C} and $B_i^{\mathrm{C}}(t)$, respectively. However, finding the exact critical instant is a challenge in multiprocessor systems, therefore an upper bound for such an instant is calculated. Following this, a notion of upper bound on the critical head and critical body are introduced and identified by $\widehat{H_i^{\mathrm{C}}}$ and $\widehat{B_i^{\mathrm{C}}}$, respectively. As a result, the worst-case response time of a task $\tau_i \in \mathcal{T}^{\mathrm{m}}$ is bounded by the smallest solution of the equation if there exist one set of servers belonging to one application, where each server has the highest priority on a core.

$$t \leq \widehat{H_i^{\mathrm{C}}} + \widehat{B_i^{\mathrm{C}}}(t), \tag{4}$$

where, $\widehat{H_i^{\mathrm{C}}}$ and $\widehat{B_i^{\mathrm{C}}}(t)$ are calculated as follows:

$$\widehat{H_i^{\mathrm{C}}} = T_s - C_s^{\min}, \tag{5}$$

where, $C_s^{\min} = \min_{\forall k:1 \leq k \leq m} C_{P_k}$ is the lowest capacity among all servers and T_s is the server replenishment period.

$$\widehat{B_i^{\mathrm{C}}}(t) = R_{\mathrm{HL/i}}(t) + R_{i/\mathrm{HL}}(t), \tag{6}$$

where $R_{HL/i}(t)$ denotes the time needed to process the workload of tasks with higher and lower priority than that of task τ_i and $R_{i/HL}(t)$ is the time needed to process τ_i itself after the higher and lower priority workload is finished [29].

B. Spin-Based Resource Sharing under Partitioned Scheduling

Under the spin-based resource sharing approach when a task spins, it spins with a priority higher than any priority level on the core, therefore, a task becomes non-preemptive while spinning. This protocol is similar to MSRP [17] and FMLP for short resources [9], [10]. Below we will briefly recapitulate the blocking terms that occur under this protocol.

Local blocking due to local resources incurred to a task $\tau_i \in \mathcal{T}_{P_k}^{\mathrm{nm}}$ is upper bounded as follows:

$$B_i^{\mathrm{L}} = \max_{\substack{\forall j,l:\rho_j < \rho_i \wedge\ \tau_i,\tau_j \in \tau_{P_k} \\ \wedge\ R_l \in \mathcal{RS}_j^{\mathrm{L}} \wedge\ \rho_i \leq ceil_{P_k}(R_l)}} \{Cs_{j,l}\}. \tag{7}$$

Local blocking due to global resources incurred to a task $\tau_i \in \mathcal{T}_{P_k}^{\mathrm{nm}}$ is upper bounded as follows:

$$B_i^{\mathrm{G}} = \max_{\substack{\forall j,q:\rho_j < \rho_i \wedge \tau_i,\tau_j \in \mathcal{T}_{P_k}^{\mathrm{nm}} \\ \wedge R_q \in \mathcal{RS}_j^{\mathrm{G}}}} \{Cs_{j,q} + spin_{P_k,q}\}. \tag{8}$$

The total local blocking that is incurred to a task τ_i is denoted by B_i and is calculated as follows:

$$B_i = \max\{B_i^{\mathrm{L}}, B_i^{\mathrm{G}}\}. \tag{9}$$

$spin_{P_k,q}$ (see Definition 4) is upper bounded as follows:

$$spin_{P_k,q} = \sum_{\forall P_r \neq P_k} \max_{\forall \tau_j \in \mathcal{T}_{P_r,q}} Cs_{j,q}. \tag{10}$$

$spin_i$ (see Definition 5) is calculated as follows:

$$spin_i = \sum_{\substack{\forall q: R_q \in \mathcal{RS}_i^{\mathrm{G}} \\ \wedge \tau_i \in \mathcal{T}_{P_k}^{\mathrm{nm}}}} n_{i,q}^{\mathrm{G}} \times spin_{P_k,q}. \tag{11}$$

The critical section length of a task and consequently its execution time will be increased by spinning. Therefore, the actual execution time of a task τ_i is denoted by $\acute{C}_i$ and is calculated as follows:

$$\acute{C}_i = C_i + spin_i. \tag{12}$$

V. Blocking Terms of Non-Migrating Tasks

In this section we present the blocking bounds incurred to non-migrating tasks. First, we show by means of Lemma 3 that according to our provided rules, the property of a spin-based approach such as MSRP, where only one task at any time can have a pending request for a global resource, is also hold here. However, when calculating the maximum spin-lock time of a global resource by a non-migrating task τ_i, it is realized that the resource may be in use on a different core by either a non-migrating or a migrating task. Therefore, (10) cannot be used anymore and adjustment to the equation is needed which is presented in Lemma 4. We first present Lemmas 1 and 2 which are essential for the proof of Lemma 3.

Lemma 1. *Only one migrating task can use the overrun budget of a server on a core (see Definition 6).*

Proof: It is immediately inferred from Rules 10, 11 and 16. ■

Lemma 2. *A migrating task that issues a request for a resource does not get preempted on the core on which it has issued the request, until it releases the resource.*

Proof: According to Rule 9, a migrating task that requests a resource becomes non-preemptive from other migrating task's point of view. Note that the server has the highest priority on the core according to the system model assumption, thus the server that is running the migrating task inside cannot get preempted by arrival of any non-migrating task. Therefore, the only possibility for a migrating task with that has issued a resource request to be preempted on the core is the depletion of the server budget. However, based on the construction of Definition 6 and having in mind Rules 10 and 11, it is guarantee that the server budget is enough until any migrating task that has requested a resource remain on the core until it releases its resource. Moreover, according to Lemma 1 no migrating task can use the remaining of the overrun budget that has already been used by another migrating task which removes the possibility of the depletion of that budget while a migrating task is consuming it. This finishes the proof. ■

Lemma 3. *In our hybrid system model, only one task on any core can have a pending request on a global resource at any time.*

Proof: We investigate the lemma for both non-migrating and migrating tasks, separately. According to Rule 6 when a non-migrating task on a processor P_k issues a request for a global resource, it becomes non-preemptive on P_k until it releases the resource. Therefore, no other non-migrating task as well as the server on that core can preempt it. Moreover, when a migrating task issues a resource request, according to Lemma 2 it cannot be preempted on the core where it issued the request until it accesses and releases the resource. As a result, if either a non-migrating task issues a resource request on a processor, or a migrating task does, it cannot be preempted on that core until the task releases the resource. This finishes the proof. ■

Lemma 4. *For a task $\tau_i \in \mathcal{T}^{\text{nm}}_{P_k}$, $spin_{P_k,q}$ (see Definition 4) is upper bounded as follows:*

$$spin_{P_k,q} = \sum_{\forall P_r \neq P_k} \max(\max_{\substack{\forall \tau_j \in \mathcal{T}_{P_r} \\ \wedge R_q \in \mathcal{RS}^{\text{G}}_j}} Cs_{j,q}, \max_{\substack{\forall \tau_h \in \mathcal{T}^{\text{m}} \\ \wedge R_q \in \mathcal{RS}^{\text{G}}_h}} Cs_{h,q}). \tag{13}$$

Proof: According to Rule 6, when a non-migrating task on a processor P_k issues a request for a global resource R_q which is hold by another task, it spins non-preemptively on P_k. Therefore, it can be delayed to access the resource, only by tasks using the same resource on a different processor. The maximum waiting time of any non-migrating task for a resource R_q on a processor P_k is when all tasks that use this resource have requested it on other cores earlier than the task on P_k and put their requests ahead of this task in the FIFO queue. Moreover, according to Lemma 3, only one task can have a pending request for a global resource at any time on any processor. This means that, when a non-migrating task on P_k requests a resource R_q, only one task's critical section on R_q from any other core can delay the task on P_k. On the other hand, the resource might be used by either a non-migrating task or a migrating task on a different core than P_k. As a result, under the worst-case scenario, a request on R_q by a non-migrating task on a processor P_k is delayed by the longest critical section on R_q from all other processors than P_k by either a non-migrating task or a migrating task. This finishes the proof. ■

Spin-delay time of a non-migrating task τ_i, i.e., $spin_i$ (see Definition 5), is calculated according to (11) by using (13) for $spin_{P_k,q}$. Similar to spin-based approaches such as MSRP, the spin-delay time is incorporated in the worst-case execution time of a non-migrating task as an inflation according to (12). Since a non-migrating task will only experience local blocking from the critical section of the lower priority non-migrating tasks, thus the local blocking due to local and global resources and the total local blocking incurred to a task $\tau_i \in \mathcal{T}^{\text{nm}}_{P_k}$ are calculated according to (7), (8) and (9), respectively, where $spin_{P_k,q}$ in (8) is calculated according to (13) and not (10) anymore. Note that the schedulability of non-migrating tasks takes the indirect blocking effect of migrating tasks into account. This is reflected in the spin-delay for a specific resource term as presented in (13) which is incorporated in the schedulability of a non-migrating task τ_i in the: (*i*) LBG term (8), as part of the blocking term B_i (9) and, (*ii*) worst-case execution time of higher priority tasks as presented in (12), as a part of interference.

VI. Blocking Terms of Migrating Tasks

In this section we provide the blocking bounds of migrating tasks that is due to resource sharing of both non-migrating tasks as well as migrating tasks.

A. Blocking by Non-Migrating Tasks

By viewing a server as a task that is scheduled along with non-migrating tasks on the same core, similar to a normal (non-migrating) task, the server may also experience blocking due to non-migrating tasks requesting resources. To calculate the maximum delay incurred to a server due to non-migrating tasks sharing resources, we present the following lemmas by utilizing the blocking terms presented in Section IV.

Lemma 5. *No delay can be incurred to a server from non-migrating tasks due to local resource access.*

Proof: This is inferred from the fact that each server has the highest priority ($\rho^{\max}_{P_k}$) on each core and is viewed as a task that shares no local resources with non-migrating tasks. Thus, according to Definition 3 no local resource access by non-migrating tasks can increase the priority of a non-migrating task higher than the priority of a server on the related core. ■

Lemma 6. *We denote the maximum blocking incurred to any server S_{P_k} due to non-migrating tasks on P_k accessing global resources as $B^{\text{G}}_{S_{P_k}}$ which is calculated as follows:*

$$B^{\text{G}}_{S_{P_k}} = \max_{\substack{\forall j,q: \tau_j \in \mathcal{T}^{\text{nm}}_{P_k} \\ \wedge R_q \in \mathcal{RS}^{\text{G}}_j}} \{Cs_{j,q} + spin_{P_k,q}\}. \tag{14}$$

Proof: It immediately follows from (8) and the fact that a server is viewed as a task which according to our system model has the highest priority ($\rho^{\max}_{P_k}$) on the core. ■

Total Server Delay. Followed by Lemmas 5 and 6, the maximum incurred delay to a server S_{P_k} on a processor P_k which has the highest priority ($\rho^{\max}_{P_k}$) compared to any non-migrating task on P_k is calculated as follows.

$$\delta_{S_{P_k}} = B^{\text{G}}_{S_{P_k}}. \tag{15}$$

Since migrating tasks are scheduled within servers, they will also experience the same delay incurred to the server. We show in Section VII how this blocking delay is incorporated in the response time of a migrating task.

B. Blocking By Migrating Tasks

A migrating task that is scheduled globally within a set of servers may experience blocking due to other migrating tasks requesting resources. A migrating task may experience two types of blocking incurred by other migrating tasks: (*i*) spin-lock time for a resource since another migrating task is holding it (see Definition 4), and (*ii*) blocking incurred by migrating tasks with lower priority when their priority is boosted due to requesting a resource (Rule 9). To calculate the maximum incurred blocking to a task due to case (*i*) and case (*ii*), we present Lemmas 7 to 10.

Lemma 7. *For a task $\tau_i \in \mathcal{T}^{\mathrm{m}}$, $spin_{i,q}$ (see Definition 4) is upper bounded as follows:*

$$spin_{i,q} = \max_{\substack{\forall P_k \\ \wedge C_{P_k} \neq 0}} \Big(\sum_{\forall P_r \neq P_k} \max\big(\max_{\substack{\forall \tau_j \in \mathcal{T}_{P_r} \\ \wedge R_q \in \mathcal{RS}_j^{\mathrm{G}}}} Cs_{j,q}, \max_{\substack{\forall \tau_h \in \mathcal{T}^{\mathrm{m}} \\ \wedge \tau_h \neq \tau_i \\ \wedge R_q \in \mathcal{RS}_h^{\mathrm{G}}}} Cs_{h,q} \big) \Big). \tag{16}$$

Proof: According to Lemma 2, a migrating task τ_i that issues a request for a resource, does not get preempted on the core on which it has issued the request until it accesses and releases the resource. Therefore, the task's access to its resource can be delayed only by tasks using the same resource on a different processor. Similar to Lemma 4, the maximum waiting time of τ_i for a request on R_q that is issued on P_k is when all requests on cores other than P_k are served in a FIFO manner earlier than τ_i's request. Similarly, according to Lemma 3, only one task from any other processor can have a pending request on a global resource at any time. Thus, the maximum delay to τ_i for a request on R_q that is issued on P_k is equal to (13). However, since τ_i is scheduled globally within the servers, different jobs of τ_i may be scheduled in any server on any processor in the system. Therefore, to account for maximum access delay to τ_i for R_q, we find the maximum of such delay incurred to τ_i assuming it may issue its request on any possible processor. This is shown by (16). ■

Similar to (11), the maximum delay that a migrating task may experience for all its resource requests, i.e., $spin_i$, is calculated as follows:

$$spin_i = \sum_{\substack{\forall q: R_q \in \mathcal{RS}_i^{\mathrm{G}} \\ \wedge \tau_i \in \mathcal{T}_{P_k}^{\mathrm{nm}}}} n_{i,q}^{\mathrm{G}} \times spin_{i,q}. \tag{17}$$

Similar to non-migrating tasks, the worst-case execution time of migrating tasks are also inflated by the spin-delay time which is calculated according to (12). The reason is that when a migrating task spins, it consumes the budget of the server, as a result the spinning time is treated similar to the task's execution cost.

Lemma 8. *Any job of a migrating task can be blocked at most once by lower priority migrating tasks for every server period during its execution.*

Proof: A migrating task τ_i can get blocked by a lower priority task when the lower priority task is non-preemptive (see Rule 9). We divide the execution interval of τ_i into two sub-intervals: (*i*) from the time when a job of τ_i arrives for the first time until the first replenishment period, and (*ii*) any upcoming server period where the execution of the job of τ_i is still not finished, until τ_i finishes its execution. We discuss the worst-case blocking scenario incurred to τ_i separately under each case. In the first case, under a worst-case scenario, when τ_i arrives, all tasks with priority lower than τ_i are non-preemptive on the cores where servers are active (i.e., the server has not been preempted on the core and it has normal budget left). Thus, τ_i is once blocked under such situation. However, as soon as any of such lower priority tasks become preemptive, τ_i can preempt them and run. Under the second case, let us assume τ_i starts executing in a server at time t_1. Now, let us assume at time $t_2 \neq t_1$ the budget of the server where τ_i is executing is depleted. However slightly before τ_2 on all other cores where the server is active, a lower priority migrating task is non-preemptable (similar to case (*i*)). Thus, τ_i will experience again a blocking by lower priority migrating tasks. The scenario under case (*ii*) can happen in every server period that τ_i's execution is not finished. ■

Lemma 9. *The maximum amount of blocking incurred to any job of a migrating task τ_i per server period is denoted as non-preemptive blocking of τ_i in a server period and presented by NPB_i^{sp} which is upper bounded as follows:*

$$NPB_i^{\mathrm{sp}} = \max_{\substack{\forall j,q: \rho_j < \rho_i, R_q \in \mathcal{RS}_j \\ \wedge R_q \in \mathcal{RS}_i \wedge \tau_i, \tau_j \in \mathcal{T}^{\mathrm{m}}}} \{Cs_{j,q} + spin_{i,q}\}. \tag{18}$$

Proof: A task with priority lower than that of τ_i can become non-preemptive when either it is spinning or executing a critical section (see Rule 9). The worst-case blocking scenario for τ_i in a server period happens when all tasks with priority lower than τ_i are spinning for a resource on the cores where servers are active (i.e., the server has not been preempted on the core and it has normal budget left) since under such scenario, τ_i will experience both a spin-delay as well as an access delay of a lower priority migrating task. Such a scenario is imaginable where one (or several) non-migrating tasks are holding those migrating tasks' requested resources on the remaining cores. Moreover, according to Lemma 8, any job of a migrating task is blocked at most once in each server period. Thus, τ_i will experience at most one such delay from any lower priority migrating task in a server period. This finishes the proof. ■

Lemma 10. *Total blocking incurred to a migrating task τ_i during its execution due to non-preemptable execution of lower priority migrating tasks is denoted by $NPB_i^{\mathrm{total}}(t)$ which is upper bounded as follows:*

$$NPB_i^{\mathrm{total}}(t) = NPB_i^{\mathrm{sp}} \times (\left\lceil \frac{t}{T_s} \right\rceil + 1). \tag{19}$$

Proof: It is immediately inferred from Lemmas 8 and 9 considering that τ_i experiences NPB_i^{sp} delay once it arrives and every time its execution is deferred to the next server period until it is finished. ■

Later in Section VII we show how $NPB_i^{\mathrm{total}}(t)$ and $\delta_{S_{P_k}}$ terms are incorporated in the response time of a migrating task.

VII. Response Time Analysis

The schedulability analysis of non-migrating tasks is evaluated based on the classical response time analysis [13].

Previously, in [29], the response time of a task τ_i, that is processed by a set of servers which schedule a set of independent tasks is presented which we also briefly presented it in Section IV-A (see Appendix C for complete equations). In this section, since resource sharing is enabled, new parameters are added to the response time driven in [29]. A migrating task may experience both the same delay that is incurred to the server by non-migrating tasks and the delay that is caused by other migrating tasks scheduled within the servers. In the following we show how these delays are incorporated in the response time of a task processed by servers presented in [29].

We showed in Section VI-A that by viewing the server on a core as a task which has the highest priority, the server may experience delay from non-migrating tasks on the same core that access resources (see (15)). This means that a migrating task executing inside the server will also experience this delay. However, such delay to τ_i should be accounted once and only in the last server period where τ_i finishes. This is due to the fact that non-migrating tasks do not consume the server budget and in the worst case the delay caused by these tasks will only defer the execution of server. This is illustrated in Figure 1. As a result, it is enough to add this delay to the total response time of τ_i, once. Since a migrating task τ_i is scheduled globally, it may be scheduled in any of the servers. As a result, the maximum such delay which τ_i may experience, is calculated by finding the largest delay imposed to any server. This is presented by the last term in (20).

Besides this delay, a migrating task may also experience an extra delay due to non-preemptive blocking by migrating tasks while it is scheduled within the servers (see (19)). Note that, as mentioned in Section VI-A, the spin-delay of a migrating task presented by (17), is treated as part of the execution time of the task, and thus is incorporated in the worst-case execution time of the task.

As a result, the response time of a migrating task τ_i denoted as WR_i^{m} presented in (4) is updated by blocking delays as below, where, WR_i^{m} is the smallest solution to the following equation.

$$t \le \widehat{H_i^{\mathrm{C}}} + \widehat{B_i^{\mathrm{C}}}(t) + NPB_i^{\mathrm{total}}(t) + \max_{\forall P_k \wedge \tau_i \in \mathcal{T}_{A_a}} \delta_{S_{P_k},A_a}. \tag{20}$$

Figure 1 illustrates the response time interval of a job of a task τ_i. In [29], it has been shown that the worst-case arrival scenario for a job of a task τ_i is when τ_i arrives slightly after the minimum budget among the servers is depleted. Under our presented model where tasks share resources, as described in Lemma 9, a task may experience a non-preemptive blocking when it arrives. Therefore, the worst-case arrival scenario of τ_i is modified as illustrated in Figure 1. Note that, the total non-preemptive blocking that can be incurred to τ_i in its response time interval, i.e., $NPB_i^{\mathrm{total}}(t)$ in (20), is the collection of NPB_i^{sp} delays in each server period as presented in (19).

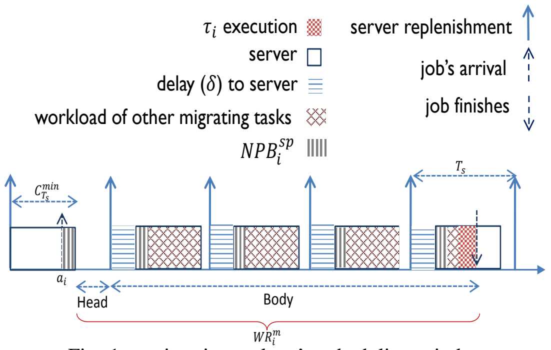

Fig. 1. migrating task τ_i's scheduling window

VIII. System Schedulability Steps

To determine the schedulability of the critical and non-critical applications, the following steps are performed. Firstly, the partitioning and schedulability of the critical application is determined. Secondly, it is checked whether or not the non-critical application can be added to the platform without jeopardizing the guarantees provided to the critical application. This check is based on the anticipated access times of the migrating tasks to shared resources. Thirdly, the total budget of the server on each core (see Definition 6) is determined based on the minimum slack among the non-migrating tasks on the core. To find the budget on a core, a similar algorithm as in [21] is used, where we incorporate the blocking terms in the demand-bound function. Fourthly, the normal budget of each server is validated to be non-zero by using (3). Finally, the schedulability of the non-critical application is determined.

Although the critical application will inherently experience interference from the non-critical application, its predictability is guaranteed through a resource reservation technique for the core combined with dedicated means to prevent resource access-time overruns of the non-critical application.

IX. Conclusion and Future Work

In this paper, we enable inter-application resource sharing in a hybrid partitioned/global scheduling framework for multiprocessors. We extended existing synchronization protocols based on FIFO-queues and spin-based techniques for such a hybrid framework. Our resource-sharing approach preserves existing properties of spin-based protocols, such as bounding the FIFO queue size to the number of cores. We provided the blocking bounds under our presented protocol and incorporated them in the existing response time analysis provided for this framework. As future work we plan to improve the resource-sharing approach for such a hybrid scheduling framework by tightening the blocking bounds and to provide quantitative evaluation results.

References

[1] AUTOSAR release 4.0. 2012, http://www.autosar.org.

[2] S. Afshar, M. Behnam, R. Bril, and T. Nolte. Flexible spin-lock model for resource sharing in multiprocessor real-time systems. In 9^{th} *IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 41–51, June 2014.

[3] S. Afshar, M. Behnam, R. Bril, and T. Nolte. Resource sharing under global scheduling with partial processor bandwidth. In 10^{th} *IEEE International Symposium on Industrial Embedded Systems (SIES)*, June 2015.

[4] S. Afshar, N. M. Khalilzad, F. Nemati, and T. Nolte. Resource sharing among prioritized real-time applications on multiprocessors. In 6^{th} *International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*, Dec. 2013.

[5] M. Asberg, T. Nolte, and M. Behnam. Resource sharing using the rollback mechanism in hierarchically scheduled real-time open systems. In 19^{th} *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 129–140, April 2013.

[6] T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, 1991.

[7] M. Behnam, I. Shin, T. Nolte, and M. Nolin. An overrun method to support composition of semi-independent real-time components. In 32^{nd} *Annual IEEE International Computer Software and Applications (COMPSAC)*, pages 1347–1352, July 2008.

[8] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In 28^{th} *IEEE International Real-Time Systems Symposium (RTSS)*, Dec. 2007.

[9] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In 13^{th} *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 47–56, Aug. 2007.

[10] B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS^{RT}. In 14^{th} *IEEE Intl. Conf. on Embedded and Real-Time Computing Sys. and Applications (RTCSA)*, pages 185–194, Aug. 2008.

[11] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In 31^{st} *IEEE Real-Time Systems Symposium (RTSS)*, pages 49–60, Dec. 2010.

[12] B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: mutex, reader-writer, and k-exclusion locks. In 9^{th} *IEEE/ACM Intl. Conference on Embedded Software (EMSOFT)*, pages 69–78, Oct. 2011.

[13] G. C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, Oct. 2004.

[14] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In 19^{th} *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 247–258, July 2007.

[15] R. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In 27^{th} *IEEE InternationalReal-Time Systems Symposium (RTSS)*, pages 257–270, Dec 2006.

[16] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In 30^{th} *IEEE Real-Time Systems Symposium (RTSS)*, pages 377–386, Dec. 2009.

[17] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. In 9^{th} *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 189–198, May 2003.

[18] T. M. Ghazalie and T. P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Syst.*, 9(1):31–67, July 1995.

[19] K.-Y. Lam, K.-K. Cheung, and J.-Y. Ng. A conditional abortable priority ceiling protocol for real-time systems with mixed tasks. In 9^{th} *Euromicro Workshop on Real-Time Systems*, pages 102–109, Jun 1997.

[20] G. Lipari and E. Bini. A framework for hierarchical scheduling on multiprocessors: From application requirements to run-time allocation. In 31^{st} *IEEE Real-Time Systems Symposium (RTSS)*, Nov 2010.

[21] M. Liu, M. Behnam, S. Kato, and T. Nolte. A server-based approach for overrun management in multi-core real-time systems. In 19^{th} *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2014.

[22] F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In 23^{rd} *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 251–261, July 2011.

[23] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[24] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In 19^{th} *Real-Time Systems Symposium (RTSS)*, pages 259–269, Dec. 1988.

[25] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sep. 1990.

[26] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In 20^{th} *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 181 –190, July 2008.

[27] H. Takada and K. Sakamura. Real-time synchronization protocols with abortable critical sections. In 1^{st} *International Workshop on Real-time Computing Systems and Application*, pages 48–52, 1994.

[28] H. Zhu, S. Goddard, and M. Dwyer. Response time analysis of hierarchical scheduling: The synchronized deferrable servers approach. In *University of Nebraska-Lincoln, Tech. Rep., 2011. [Online]. Available: http://ponca.unl.edu/facdb/csefacdb/TechReportArchive/TR-UNL-CSE-2011-0006.pdf*.

[29] H. Zhu, S. Goddard, and M. Dwyer. Response time analysis of hierarchical scheduling: The synchronized deferrable servers approach. In 32^{nd} *IEEE Real-Time Systems Symposium (RTSS)*, pages 239–248, Nov. 2011.

APPENDIX A
PROCESSOR SLACK

In this section, we describe how to find the slack on a processor in order to be assigned to servers as their budget. In order to find the slack on a processor P_k, the minimum slack among tasks with priority lower than that of the server is found and not among the higher priority tasks. This is due to the fact that server can only cause interference to tasks with lower priority than itself. However, we still need to make sure that the server itself is schedulable. For this purpose, we assume the server as a task (denoted in the algorithm as τ_s with $C_s = 0$ with priority ρ_s) where its inter arrival time is equal to the server period (lines 3 to 5 in Algorithm 1). Therefore, the minimum slack among all lower priority tasks as well as τ_s, specifies the slack on P_k (line 15 in Algorithm 1). Slack of a task is specified according to Algorithm 2. The calculated slack of the task is then divided by $\frac{FindSlack(\tau_i)}{\lceil T_i/T_s \rceil+1}$ to assign the budget for one server period.

Algorithm 1 Processor P_k Budget Assigning Algorithm

1: Initialize $ProcSlack$
2: Initialize $TaskList \leftarrow \oslash$
3: Initialize τ_s:$\{C_s, T_s\}$
4: $C_s \leftarrow 0$
5: **for all** $\tau_i \in \mathcal{T}_{\mathrm{P_k}}$ **do**
6:     **if** $\rho_i < \rho_s$ **then**
7:         τ_i added to $TaskList$
8:     **end if**
9: **end for**
10: τ_s added to $TaskList$
11: **for all** $\tau_i \in TaskList$ **do**
12:     $slack_i = \frac{FindSlack(\tau_i)}{\lceil T_i/T_s \rceil+1}$
13: **end for**
14: $ProcSlack = \min\limits_{\forall \tau_j \in TaskList} slack_j$

To find the slack of a task $\tau_i \in \mathcal{T}_{\mathrm{P_k}}$, the difference between the incurred load to τ_i and the processor supply is calculated at a set of check points in time (line 31 in Algorithm 2). The check points are multiplications of all higher priority tasks' period considered until the task τ_i's period (lines 10 to 19 in Algorithm 2). The task τ_i's slack is the maximum value for differences between the check points and the incurred load in that point in time (lines 32 to 34 in Algorithm 2). The algorithm returns a positive value which is the maximum slack provided by task τ_i, otherwise it returns -1, if the task has a negative slack which means that the task misses its deadline. Note that the $\acute{C}_i$ in line 28 and the B_i in line 30 are calculated according to Section V.

APPENDIX B
NOTATIONS

Here are the notations that have been used in this paper:
P_k: processor k.
τ_i: task i.
C_i: worst-case execution time of τ_i.
T_i: minimum inter arrival time of τ_i.
D_i: relative deadline of τ_i.
a_i: arrival time of any job instance of τ_i.
d_i: absolute deadline of any job of τ_i.
ρ_i: priority of τ_i.
$\mathcal{T}_{P_k}^{\mathrm{nm}}$: set of non-migrating tasks (tasks of the critical application) assigned to P_k.
S_{P_k,A_a}: server related to application A_a dedicated to core P_k.
C_{P_k}: capacity of S_{P_k}.
ρ_{S_k}: priority of S_{P_k}.

Algorithm 2 FindSlack(τ_i)

Initialize $checkPoint \leftarrow 0$
Initialize $checkPointList \leftarrow \oslash$
Initialize $hpTaskList \leftarrow \oslash$
$\tau_i \in \mathcal{T}_{\mathrm{P_k}}$
for all $\tau_h \in \mathcal{T}_{\mathrm{P_k}}$ **do**
  if $\rho_h > \rho_i$ **then**
    τ_h added to $hpTaskList$
  end if
end for
for all $\tau_h \in hpTaskList$ **do**
  $k \leftarrow 1$
  $checkPoint = k \times T_h$
  while $checkPoint < D_i$ **do**
    if $checkPoint \notin checkPointList$ **then**
      $checkPoint$ added to $checkPointList$
    end if
    $k++$
  end while
end for
D_i added to $checkPointList$
$load \leftarrow 0$
$maxTaskSlack \leftarrow 0$
for all $t \in checkPointList$ **do**
  $hpInterference \leftarrow 0$
  $slack \leftarrow 0$
  for all $\tau_h \in hpTaskList$ **do**
    $hpInterference += \left\lceil \frac{t}{T_h} \right\rceil \times \acute{C}_h$
  end for
  $load = \acute{C}_i + hpInterference + B_i$
  $slack = t - load$
  if $slack > maxTaskSlack$ **then**
    $maxTaskSlack \leftarrow slack$
  end if
end for
if $maxTaskSlack < 0$ **then**
  return -1
end if
return $maxTaskSlack$

T_s: server replenishment period.
$\mathcal{T}^{\mathrm{m}}$: set of migrating tasks.
R_q: resource q.
$\mathcal{R}^{\mathrm{L}}_{P_k}$: set of local resources accessed by tasks on P_k.
$\mathcal{RS}_i$: set of resources accessed by jobs of τ_i.
$\mathcal{RS}^{\mathrm{L}}_i$: set of local resources accessed by jobs of τ_i.
$\mathcal{RS}^{\mathrm{G}}_i$: set of global resources accessed by jobs of τ_i.
$Cs_{i,q}$: worst-case execution time in all τ_i's requests on R_q.
n^{G}_i: maximum number of τ_i's global requests.
$n^{\mathrm{G}}_{i,q}$: maximum number of requests of τ_i for global resource R_q.
$n_{i,q}$: number of τ_i's requests on R_q.
WR^{nm}_i: worst-case response time of a non-migrating task τ_i.
WR^{m}_i: worst-case response time of a migrating task τ_i.

Appendix C
Higher and Lower Priority Workload Recap

$R_{HL/i}(t)$ which is the upper bound of higher and lower priority workload that is processed before a task $\tau_i \in \mathcal{T}_{A_a}$ is calculated for $t = WR^{\mathrm{m}}_i$) and is as follows:

$$R_{\mathrm{HL/i}}(t) = (\lceil \frac{W^i_{\mathrm{HL}}(t)}{\sum_{k=1}^{m} C_{P_k}} \rceil - 1).T_s + t^{\mathrm{HL}}_{\mathrm{res}}(t), \tag{21}$$

where, $W^i_{HL}(WR^{\mathrm{m}}_i)$ is specified according to (27) and t^{HL}_{res} is calculated as below:

$$t^{\mathrm{HL}}_{\mathrm{res}} = \begin{cases} \dfrac{W^{\mathrm{HL}}_{\mathrm{res}}(t)}{m} & \text{if } W^{\mathrm{HL}}_{\mathrm{res}}(t) \leq \delta(m) \\ C_{P_{(k+1)}} + \\ \dfrac{W^{\mathrm{HL}}_{\mathrm{res}}(t) - \delta(k+1)}{k} & \text{if } \delta(k+1) < W^{\mathrm{HL}}_{\mathrm{res}}(t) \leq \delta(k), \\ & \forall \text{k}: 1 \leq k \leq m-1 \end{cases}$$

$$W^{\mathrm{HL}}_{\mathrm{res}}(t) = W^i_{\mathrm{HL}}(t) - (\lceil \frac{W^i_{\mathrm{HL}}(t)}{\sum_{k=1}^{m} C_{P_k}} \rceil - 1).\sum_{k=1}^{m} C_{P_k}. \tag{22}$$

$$\forall k: 1 \leq k \leq m: \delta(k) = \sum_{p=k}^{m} C_{P_p} + C_{P_k}.(k-1). \tag{23}$$

$R_{i/HL}$ which denotes the upper bound of the needed time to process τ_i after the higher and lower priority workload is finished is calculated as follows:

$$R_{i/\mathrm{HL}} = \begin{cases} C_i & \text{if } C^{\mathrm{rmn,i}}_s \geq C_i \\ T_s - t^{\mathrm{HL}}_{\mathrm{res}}(t) + CRP_i.T_s + C_i - \\ CRP_i(t).min(\sum_{k=1}^{m} C_{P_k}, T_s) & \text{otherwise} \end{cases} \tag{24}$$

where,

$$CRP_i(t) = \lceil \frac{C_i - C_{\mathrm{rmn}}(t)}{\sum_{k=1}^{m} C_{P_k}} \rceil - 1, \tag{25}$$

$$C_{\mathrm{rmn}}(t) = min(\sum_{k=1}^{m} C_{P_k} - W^{\mathrm{HL}}_{\mathrm{res}}(t), T_s - t^{\mathrm{HL}}_{\mathrm{res}})(t). \tag{26}$$

$$W^i_{\mathrm{HL}}(t) = W^i_{\mathrm{HP}}(t) + W^i_{\mathrm{LP}}(t), \tag{27}$$

where $W^i_{HP}(WR^{\mathrm{m}}_i)$ and $W^i_{LP}(WR^{\mathrm{m}}_i)$ presents the upper bounds of the workload of lower and higher priority tasks in the interval of WR^{m}_i as follows.

Zhu et al. [29] showed that the workload of the tasks with lower priority than that of task τ_i also affect by the response time of τ_i and is calculated as follows:

$$W^i_{\mathrm{LP}}(t) = min(\sum_{j<i} RW^i_j(t), CCL_i(t)), \tag{28}$$

where for $t = WR^{\mathrm{m}}_i$ $CCL_i(t)$ can be bounded from above as $CCL_i(WR^{\mathrm{m}}_i) = (m-1).C_i$ and $RW^k_j(t)$ is calculated according to (30).

The workload of higher priority tasks than that of task τ_i ($W^i_{\mathrm{HP}}(WR^{\mathrm{m}}_i)$) is calculated for $t = WR^{\mathrm{m}}_i$ as follows:

$$W^i_{\mathrm{HP}}(t) = \sum_{j>i} RW^i_j(t), \tag{29}$$

where $RW^i_j(WR^{\mathrm{m}}_i)$ denotes the upper bound of the requested workload of a task τ_j in the interval of WR^{m}_i and is calculated similar to [8] presented in (30).

$$\begin{aligned} &\forall j \neq i: \\ &RW^i_j(t) = N_j(t).C_j + min(C_j, t + D_j - C_j - N_j(t).T_j), \end{aligned} \tag{30}$$

where $N_j(t) = \left\lfloor \frac{t + D_j - C_j}{T_j} \right\rfloor$.