Resource Sharing Under Global Scheduling with Partial Processor Bandwidth

Sara Afshar[1], Moris Behnam[1], Reinder J. Bril[1,2], Thomas Nolte[1]
[1]Mälardalen University, Västerås, Sweden
[2]Technische Universiteit Eindhoven, Eindhoven, Netherlands
Email: {sara.afshar, moris.behnam, reinder.j.bril, thomas.nolte}@mdh.se, r.j.bril@tue.nl

Abstract—**Resource efficient approaches are of great importance for resource constrained embedded systems. In this paper, we present an approach targeting systems where tasks of a critical application are partitioned on a multi-core platform and by using resource reservation techniques, the remaining bandwidth capacity on each core is utilized for one or a set of non-critical application(s). To provide a resource efficient solution and to exploit the potential parallelism of the extra applications on the multi-core processor, global scheduling is used to schedule the tasks of the non-critical applications. Recently a specific instantiation of such a system has been studied where tasks do not share resources other than the processor. In this paper, we enable semaphore-based resource sharing among tasks within critical and non-critical applications using a suspension-based synchronization protocol. Tasks of non-critical applications have partial access to the processor bandwidth. The paper provides the systems schedulability analysis where blocking due to resource sharing is bounded. Further, we perform experimental evaluations under balanced and unbalanced allocation of tasks of a critical application to cores.**

I. Introduction

With a growing interest towards replacing traditional single core[1] processors with new multi-cores as the defacto processors in embedded systems, a demand has emerged for investigating proper scheduling techniques to allow for such a migration. One major concern in this context is that embedded systems typically have a constrained amount of resources. Therefore, techniques that can enable an efficient usage of processor bandwidths are of great importance.

In general two conventional scheduling approaches exist for multiprocessor systems being *partitioned* and *global* scheduling [7], [16]. In the partitioned approach, each task is assigned to a processor at design time and it will be scheduled only on its assigned processor during run-time. However, in global scheduling, tasks are selected from one system wide unique global queue at run time and they are scheduled on any available processor (among the idle processors). In global scheduling tasks may migrate among processors whereas in the partitioned scheduling approach tasks are bound to processors. Global scheduling approaches are privileged to partitioned approaches due to their higher system utilization guarantees since they allow a task to run whenever a processor becomes available. However, global scheduling techniques introduce more overhead to the system due to potential migrations of tasks among processors.

From an industrial point of view, co-existence of multiple real-time applications on a shared multi-core platform is an efficient solution since it can provide re-usability of the independently-developed applications besides decreasing the system power consumption and costs. By transition to a multi-core architecture, these subsystems/applications which may share resources will eventually co-exist on a shared multi-core platform. These applications, may have different criticality levels. In this paper we consider two types of applications[2]: *critical* and *non-critical*.

In practice, such as automotive industry [1], critical applications are partitioned in the multiprocessor system to achieve a higher degree of predictability. Moreover, resource efficient solutions for embedded systems suggest to further utilize the remaining bandwidth from such applications on each core. To provide temporal guarantees to the critical application, served-based techniques can be applied [4], [22].

In this paper, we target a resource efficient structure where a set of critical applications are partitioned on the multiprocessor platform and the remaining capacity on each core is determined statically, and made available to a set of non-critical applications through resource reservation techniques. The critical application resides on the platform whereas other non-critical applications can be added or removed dynamically. In [4] global scheduling has been used for less critical applications. Similarly and to exploit the potential parallelism, we also use global scheduling to schedule the tasks of non-critical applications.

A specific instantiation of such a platform has been studied in [31]. However, this approach is based on a simplifying assumption that tasks do not share resources with each other, except the processor. In this work we enable semaphore-based resource sharing assuming that each critical and non-critical application uses a dedicated set of resources. We use a suspension-based resource sharing approach i.e., a task suspends whenever the resource is not available letting other tasks to execute. Note that, our focus here is handling resources that are protected by locks. We do not consider physical resources such as a bus or memory as done in [25].

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.

[2]Our main focus in this paper is to enable resource sharing and we do not focus on the solutions for more complex mixed-criticality systems where mode changing happens.

Contributions: In this paper, we enable intra-application resource sharing within critical and non-critical applications. We bound the blocking duration of tasks and derive the schedulability analysis tailored to the proposed synchronization protocol. Moreover, we perform experimental evaluations where we investigate the system schedulability using two partitioning techniques of the critical application: balanced and unbalanced.

The rest of this paper is structured as follows: Section II summarizes the existing related works in this context. Section III defines our system and resource sharing model as well as the scheduling and resource sharing rules. Section IV presents a recap of the existing analysis. Sections V and VI presents the blocking bounds and our new response time analysis based on the resource sharing parameters. Section VII presents evaluation and experimental results. Finally, Section VIII concludes.

II. Related Works

Hierarchical scheduling for multiprocessor platforms has been studied [15], [28], [21], [31]. However, these works have used the simplifying assumption that tasks are independent and that they do not share any resources other than the CPU in the system. In the context of semaphore-based resource management protocols for non-hierarchical multiprocessor schemes, a significant amount of work has been presented over the years. In the following we will briefly present the most related synchronization protocols for multiprocessor systems.

The Multiprocessor Priority Ceiling Protocol (MPCP) was introduced for partitioned systems [26]. MPCP is a variant of the Priority Ceiling Protocol (PCP) [27] for multi-core platforms. MPCP is a suspension-based protocol. The Multiprocessor Stack Resource Policy (MSRP) was introduced in [20] for partitioned systems. MSRP is an extension of the Stack Resource Policy (SRP) [6] for multiprocessors and it is a spin-based approach (i.e., a task requesting a resource busy waits and does not leave the processor until the resource becomes available).

The Flexible Multiprocessor Locking Protocol (FMLP) has been introduced in [9] under two variants for partitioned and global scheduling respectively. Later in [10] the partitioned FMLP was extended for fixed priority scheduling. A synchronization protocol called $O(m)$ Locking Protocol (OMLP) was proposed in [11] under both partitioned and global scheduling.

The Multiprocessor Synchronization Protocol for Open Systems (MSOS) is a suspension-based preemptive synchronization protocol developed for compositional independently-developed real-time applications introduced in [24]. Later in [3] MSOS was extended to applications that are assigned priority to improve the schedulability performance. In [29], different types of queue strategies have been investigated for spin-based multiprocessor systems where mixed-integer linear program (ILP) has been used to bound the maximum cumulative blocking incurred to a task.

In [12], resource sharing for cluster-based scheduling has been investigated. This approach uses a partitioning technique to bound tasks to clusters of processors and schedule tasks globally inside each cluster.

In [19], two new schedulability tests for PIP and Parallel-PCP synchronization protocols which are suspension-based approaches have been proposed. In this work, a response time schedulability analysis for both approaches which are developed under fixed-priority global scheduling has been proposed.

Unlike the system we are investigating in this paper, all aforementioned synchronization protocols assume full access to the processor bandwidth.

III. System Model

A. Task Model

The system consists of a set of applications where each application consists of a set of tasks. We denote each task τ_i by $< C_i, D_i, T_i >$, where C_i denotes the worst-case execution time, D_i denotes the relative deadline and T_i is the minimum inter-arrival time. We assume a constrained-deadline task model, i.e., $D_i \leq T_i$. We denote the priority of a task τ_i by ρ_i, where we assume $\rho_i > \rho_j$ if $i > j$. We use a_i, d_i and f_i as arrival time, the absolute deadline and the finishing time of any job instant of a task τ_i.

B. Architecture and Scheduling Strategy

Our platform model consists of m identical unit capacity processors. The system contains two types of applications, one critical application and one (or more) additional non-critical application(s). For ease of presentation, we only consider a single non-critical application. The tasks of the critical application are partitioned and also called non-migrating tasks. The tasks of the non-critical application are globally scheduled, and also called migrating tasks.

Migrating tasks are globally scheduled within a set of synchronized deferrable servers that are allocated to a set of m' processors, where $m' \leq m$ (similar to [31]). First, non-migrating tasks are partitioned across the platform and the remaining bandwidth on each core is made available for a set of migrating tasks through the use of real time server-based scheduling techniques. For the sake of presentation simplicity we assume that each core accommodates one server with the highest priority on the core similar to [31]. However, in case some tasks related to the set of partitioned tasks have tight finalization jitter constraints as they belong to a critical application, the model and accompanying analysis can simply be generalized to the case where server may have any arbitrary priority (in this case a priority lower than that of such tasks on the same core) as in [30], to remove the effects of induced jitter by the server. Moreover, similar to [30], the analysis can be extended to accommodate multiple servers per core.

A server dedicated to a processor P_k is denoted by S_{P_k}. The budget of a server S_{P_k} is denoted by C_{P_k}. The server S_{P_k} executes the pending workload of migrating tasks whenever it obtains P_k. S_{P_k} can execute the pending workload up to its allocated execution capacity C_{P_k} in each of its replenishment periods. For ease of presentation, a common replenishment period T_s is assumed for all servers, however each application can have a different replenishment period.

The priority of the non-migrating tasks are assigned according to a fixed priority algorithm (e.g. the deadline monotonic (DM) technique) on a processor. The priority of the server S_{P_k} is denoted by ρ_{P_k} which is assigned as the highest priority on P_k. The set of migrating and non-migrating tasks in the system are denoted by $\mathcal{T}^{\mathrm{m}}$ and $\mathcal{T}^{\mathrm{nm}}$, respectively. $\mathcal{T}^{\mathrm{nm}}_{P_k}$ denotes the set of non-migrating tasks allocated to a processor P_k.

The allocation technique used for partitioning of non-migrating tasks to processors is not the focus of this work. Due to the complexity of the problem, for the first step, we assume that there exist an allocation solution to partition the non-migrating tasks . Later we can take the knowledge gained from this step into consideration for partitioning as a future optimization phase.

C. Resource Sharing Parameters

Local resources are the resources that are accessed by jobs of tasks on the same processor only, whereas global resources are accessed by tasks on more than one processor. In this work we assume independent applications, i.e. we look at intra-application resource sharing, only. Therefore, the notions of local and global resources are meaningful within each application. Non-migrating tasks may use both *local* and *global* resources. We denote the set of local resources which are accessed by tasks on a processor P_k by $\mathcal{R}^{\mathrm{L}}_{P_k}$. Similarly, the set of local and global resources accessed by jobs of a task τ_i are denoted as $\mathcal{RS}^{\mathrm{L}}_i$ and $\mathcal{RS}^{\mathrm{G}}_i$, respectively. We use $\mathcal{RS}_i$ to denote the set of resources accessed by jobs of τ_i. Further, the worst-case execution time among all requests of any job of a task τ_i for a resource R_q is denoted by $Cs_{i,q}$. Finally, n^{G}_i and $n^{\mathrm{G}}_{i,q}$ denotes task $\tau_i \in \mathcal{T}^{\mathrm{nm}}_{P_k}$'s maximum number of requests for any global resource and a specific global resource R_q, respectively. By definition, we call the resources that are accessed by migrating tasks also global resources since tasks using those resources are scheduled globally. Nested access of resources is not the focus of this paper, however it can be supported by using group locks similar to the approach used in [9].

In this paper we mean by a task that is granted access to a resource that it has locked the resource. However, a task that locks a resource may not be allowed to run. When a task is allowed to run while holding a resource, then the task will access the resource. Note that since nesting of resource requests is not allowed here, therefore, deadlock is prevented.

We divide the delay that may be introduced to any task's execution due to resource sharing in two categories: *Local Blocking* and *Remote Blocking*. We mean by local blocking the priority inversion blocking [27] (*pi-blocking)* that may be incurred to a task which happens when a low priority task is scheduled while a higher priority task is ready. Remote blocking is the acquisition delay incurred to a task when the task wants to obtain a resource which is locked by a task on a remote processor (i.e. a processor other than the task's assigned processor).

D. General Definitions

Definition 1. *We denote the highest normal (original) priority on a processor P_k by $\rho^{\max}_{P_k}$.*

Definition 2. *The highest priority level among migrating tasks is denoted by $\rho^{\max}_{\mathcal{T}^{\mathrm{m}}}$ and is presented as follows:*

$$\rho^{\max}_{\mathcal{T}^{\mathrm{m}}} = \max_{\forall \tau_i \in \mathcal{T}^{\mathrm{m}}} \rho_i. \tag{1}$$

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

E. Scheduling and Resource Sharing Rules

This section presents the rules that are used to schedule tasks in our platform. Since, according to our system model, migrating tasks do not share resources with non-migrating tasks, servers (that are used to schedule migrating tasks) can be viewed as independent tasks from a uniprocessor scheduling perspective. Therefore, we model a server S_{P_k} as an independent task with execution C_{P_k} and period T_s. By means of such a view, we shall consider a two-level hierarchical scheduling. In the top level, non-migrating tasks along with the server on each core are scheduled using uniprocessor scheduling. In the second level, migrating tasks are scheduled globally within servers.

We use FIFO-based queue ordering to serve the global resource requests for both migrating and non-migrating tasks to prevent lower priority tasks to starve due to probable multiple release of higher priority tasks similar to approaches in [9] and [24], respectively. Furthermore, resource holding jobs benefit from priority boosting to hasten the release of the resource.

1) Resource Sharing Among Non-Migrating Tasks: As mentioned earlier in this section, servers can be viewed as tasks with no resources. Therefore, handling resource sharing among non-migrating tasks can be achieved by using existing resource sharing approaches for partitioned scheduling. For the sake of protocol completeness, next we briefly recapitulate the resource sharing rules of such similar approaches [9], [24].

Rule 1. *Local resources are handled by means of a uniprocessor synchronization protocol e.g. SRP or PCP.*

Rule 2. *For each global resource a FIFO-based queue is used to enqueue the tasks waiting for the related resource.*

Rule 3. *Whenever a task $\tau_i \in \mathcal{T}^{\mathrm{nm}}$ requests a global resource that is used by a task on a remote processor, it is suspended and it places its request in the related resource queue.*

Rule 4. *When a task $\tau_i \in \mathcal{T}^{\mathrm{nm}}_{P_k}$ is granted access to a global resource, its priority is boosted in an atomic operation to: $\rho^{\max}_{P_k} + \rho_i$. As a result, if multiple tasks are granted access to global resources on P_k, they will access these resources according to their priority order.*

Rule 5. *A task executes a global critical section non-preemptively until it releases the resource.*

Rule 6. *The priority of the task is changed to its normal (original) priority as soon as it finishes the global critical section where it also becomes preemptable again.*

Rule 7. *When a global resource becomes available (i.e. it is released), the task at the head of the related resource queue (if any) resumes and locks the resource.*

2) Migrating Tasks Scheduling: Next, we present the rules for scheduling the migrating tasks. By incorporating resumption we have adjusted Rule 8 to cover resource sharing as well. Rule 9 summarizes the scheduling strategy introduced in [31].

Rule 8. *Migrating tasks are scheduled among servers from a unique global priority-ordered queue based on a fixed-priority preemptive global scheduling. Migrating tasks are inserted to the queue after they have been released, preempted or resumed (since resource sharing is enabled here).*

Rule 9. *If multiple servers (belonging to the same application) are available, i.e. the server is not preempted and it has remaining capacity, the highest priority (ready) task is scheduled on the server with the largest capacity.*

Note that a migrating task running within a server may be preempted under three situations: (i) a higher priority migrating task preempts the task, (ii) the capacity of the server is depleted or (iii) the server gets preempted by a non-migrating task which is granted access to a global resource on the same core (Rule 4). In all these situations the task which is preempted may also be holding a resource. In all three aforementioned situations, the task that is been preempted, is rescheduled among the available servers according to Rule 8.

3) Intra-application Resource Sharing: Handling resource requests of the tasks within an application is similar to global resource sharing of non-migrating tasks. This implies that, Rules 2, 3, 5, 6 and 7 are valid here as well. We adjust Rule 4 for intra-application resource augmentation as follows:

Rule 10. *When a task $\tau_i \in \mathcal{T}^{\mathrm{m}}$ is granted access to a resource (all resources used by migrating tasks are by definition global), its priority is boosted in an atomic operation to: $\rho^{\max}_{\mathcal{T}^{\mathrm{m}}} + \rho_i$. As a result, if multiple tasks are granted access to global resources inside an application, they will access these resources according to their priority order.*

The intuition behind the priority boosting as a function of task's original priority in this rule is that if only one server is available to execute the tasks (e.g., all servers except one is preempted or their capacity is depleted), and multiple tasks are granted access to (different) global resources, then they will be served based on their original priority order (similar to the idea as in Rule 4).

Note that, the conclusion from Rule 4, and the fact that servers are seen as independent tasks from the non-migrating tasks point of view, result in servers to be fully preemptive by non-migrating tasks that are granted access to a global resource.

IV. Existing Approaches Recap

In this section we briefly present a recap of the server-based scheduling analysis without resource sharing presented in [31] and resource sharing for non-migrating tasks.

A. Response Time of Tasks Processed by Servers

The response time analysis for server-based scheduling assuming that tasks are independent (i.e. without any resource sharing) has been studied in [31]. 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. Later in Section VI we revisit these notions to address intra-application resource sharing where we use a new concept for body which incorporates an extra delay due to resource sharing. Moreover, a new concept called starvation period is introduced which includes the previous head interval introduced in [31]. 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 are called the critical head and the critical body denoted by H^{C}_i and $B^{\mathrm{C}}_i(t)$, respectively. However, finding the exact critical instant is a challenge in multiprocessor systems, therefore calculation of the exact H^{C}_i and $B^{\mathrm{C}}_i(t)$ is not possible. Following this, a notion of upper bound on the critical head and critical body has been introduced and identified by $\widehat{H^{\mathrm{C}}_i}$ and $\widehat{B^{\mathrm{C}}_i}$, respectively. As a result, the worst-case response time of a task $\tau_i \in \mathcal{T}_{A_a}$ which we denote as WR^{m}_i has been bounded in [31] by the smallest solution to (2). τ_i is schedulable if $WR^{\mathrm{m}}_i \le D_i$.

$$t \le \widehat{H^{\mathrm{C}}_i} + \widehat{B^{\mathrm{C}}_i}(t), \tag{2}$$

where, $\widehat{H^{\mathrm{C}}_i}$, $\widehat{B^{\mathrm{C}}_i}(t)$ is calculated as follows:

$$\widehat{H^{\mathrm{C}}_i} = T_s - C^{\min}_s. \tag{3}$$

where, $C^{\min}_s = \min_{\forall k: 1 \le k \le m} C_{P_k}$ is the lowest capacity among servers.

$$\widehat{B^{\mathrm{C}}_i}(t) = R_{\mathrm{HL/i}}(t) + R_{i/\mathrm{HL}}(t), \tag{4}$$

$R_{\mathrm{HL/i}}(t)$ denotes the time needed to process the workload of tasks with higher and lower priority than that of task τ_i and $R_{i/\mathrm{HL}}(t)$ is the time needed to process τ_i [31].

Calculation of $R_{\mathrm{HL/i}}(t)$ and $R_{i/\mathrm{HL}}(t)$ has been presented in [31] and also can be viewed in Appendix C and are calculated according to (27) and (30).

B. Partitioned Synchronization Approach

Since the servers are viewed as independent tasks from non-migrating tasks perspective, they will not add additional blocking to the non-migrating tasks. Therefore, well known existing resource sharing approaches that are suitable for partitioned scheduling can be used for handling resource sharing among non-migrating tasks such as partitioned static priority FMLP for long resources (P-SP FMLP) [9], [10] and MSOS [24], [3]. In the following, we will present the pi-blocking terms that are relevant for partitioned scheduling.

Pi-blocking that is incurred to a task $\tau_i \in \mathcal{T}^{\mathrm{nm}}_{P_k}$ due to locking local resources is denoted by B^{L}_i and is upper bounded by:

$$B^{\mathrm{L}}_i = N^{\mathrm{L}}_i(t) \times \max_{\substack{\forall j,\ell:\rho_j<\rho_i \wedge\ \tau_i,\tau_j\in\mathcal{T}^{\mathrm{nm}}_{P_k} \\ \wedge\ R_\ell\in\mathcal{RS}^{\mathrm{L}}_j \wedge\ \rho_i\leq ceil_{P_k}(R_\ell)}} \{Cs_{j,\ell}\}, \tag{5}$$

where $N^{\mathrm{L}}_i(t)$ is the maximum number of such pi-blocking incurred to τ_i which is upper bounded as follows:

$$N^{\mathrm{L}}_i = \min\{n^{\mathrm{G}}_i + 1, \sum_{\substack{\forall j:\rho_j<\rho_i \\ \wedge\tau_i,\tau_j\in\mathcal{T}^{\mathrm{nm}}_{P_k}}} ((\left\lceil \frac{T_i}{T_j}\right\rceil + 1)\times n^{\mathrm{L}}_j(\tau_i))\}, \tag{6}$$

where $n^{\mathrm{L}}_j(\tau_i)$ denotes the number of requests of a task τ_j with priority lower than that of τ_i on P_k for any local resource $R_\ell \in \mathcal{RS}^{\mathrm{L}}_j$ with $\rho_i \leq ceil_{P_k}(R_\ell)$.

Pi-blocking that is incurred to a task $\tau_i \in \mathcal{T}^{\mathrm{nm}}_{P_k}$ due to locking global resources is denoted by B^{G}_i and is upper bounded by:

$$B^{\mathrm{G}}_i = \sum_{\substack{\forall j:\rho_j<\rho_i \\ \wedge\tau_i,\tau_j\in\mathcal{T}^{\mathrm{nm}}_{P_k}}} \Big(N^{\mathrm{G}}_{i,j} \times \max_{\forall R_q\in\mathcal{RS}^{\mathrm{G}}_j}\{Cs_{j,q}\}\Big), \tag{7}$$

where $N^{\mathrm{G}}_{i,j}$ is the maximum number of such pi-blocking that τ_i may experience from a lower priority task τ_j on P_k and is upper bounded as follows:

$$N^{\mathrm{G}}_{i,j} = \min\{n^{\mathrm{G}}_i + 1, ((\left\lceil \tfrac{T_i}{T_j}\right\rceil + 1)\times n^{\mathrm{G}}_j)\}. \tag{8}$$

The duration of time that any task on a processor P_k may wait to lock a global resource R_q which is locked by tasks on remote processors is denoted as $RWT_{P_k,q}$ (resource waiting time) and is upper bounded as follows:

$$RWT_{P_k,q} = \sum_{\forall P_r\neq P_k} \sum_{\substack{\forall\tau_j\in\mathcal{T}^{\mathrm{nm}}_{P_r} \\ \wedge R_q\in\mathcal{RS}^{\mathrm{G}}_j}} RHT_{j,r,q}, \tag{9}$$

where, $RHT_{j,r,q}$ denotes the maximum resource holding time of a task τ_j on a remote processor P_r for R_q which is calculated from the time that τ_j is granted access to R_q and is upper bounded as below.

$$RHT_{j,r,q} = Cs_{j,q} + HPI_{j,r,q} + \max_{\substack{\forall t,s:\rho_t<\rho_j\wedge\tau_t\in\mathcal{T}^{\mathrm{nm}}_{P_r} \\ \wedge R_s\in\mathcal{RS}^{\mathrm{G}}_t\wedge R_s\neq R_q}} Cs_{l,s}, \tag{10}$$

where, $HPI_{j,r,q}$ denotes the interference of higher priority tasks introduced to a task $\tau_j \in \mathcal{T}_{P_r}$ due to been granted global resources other than R_q and is calculated as follows:

$$HPI_{j,r,q} = \sum_{\substack{\forall h:\rho_h>\rho_j \\ \wedge\tau_h\in\mathcal{T}^{\mathrm{nm}}_{P_r}}} \max_{\substack{\forall s:R_s\in\mathcal{RS}^{\mathrm{G}}_h \\ \wedge R_s\neq R_q}} Cs_{h,s}. \tag{11}$$

Note that, $RHT_{j,r,q}$ accounts for the global critical section of τ_j on R_q, the interference from higher priority tasks' global critical sections and one lower priority task's that access its global critical section (and become non-preemptive according to Rule 5) when τ_j is granted access to R_q.

The maximum amount of time that a task $\tau_i \in \mathcal{T}^{\mathrm{nm}}_{P_k}$ has to wait in total for all its global resource requests is denoted by RWT_i and is calculated as follows:

$$RWT_i = \sum_{\substack{\forall q:R_q\in\mathcal{RS}^{\mathrm{G}}_i \\ \wedge\tau_i\in\mathcal{T}^{\mathrm{nm}}_{P_k}}} n^{\mathrm{G}}_{i,q} \times RWT_{P_k,q}. \tag{12}$$

The total delay that is incurred to a task $\tau_i \in \mathcal{T}^{\mathrm{nm}}_{P_k}$ due to resource sharing is calculated as follows:

$$B_i = B^{\mathrm{L}}_i + B^{\mathrm{G}}_i + RWT_i, \tag{13}$$

V. Blocking Terms

In this section we investigate the effect of resource sharing by non-migrating tasks to the server on a core. Moreover, we provide blocking bounds of migrating tasks due to intra-application resource sharing.

A. Server Blocking

As mentioned in our system model, servers can be viewed as independent tasks that are scheduled along with non-migrating tasks on each core. According to Rule 4, a non-migrating task that is granted access to a global resource will get a priority higher than any task or server on the related core, and as a result it can preempt the server. By viewing a server as a task, the maximum delay to a server can be calculated using the blocking bounds presented in Section IV.

In the following, we will derive the effect of resource sharing of non-migrating tasks to the server on the same core. By viewing a server as a task with the highest priority on the core that do not share any resource with non-migrating tasks, the maximum delay incurred to a server can be calculated according to the following two lemmas.

Lemma 1. *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 on each core and is viewed as a task that share no 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 2. *We denote the maximum blocking incurred to any server S_{P_k} due to non-migrating tasks on P_k accessing global resources as $B^{\mathrm{G}}_{S_{P_k}}$ which is calculated as follows:*

$$B^{\mathrm{G}}_{S_{P_k}} = \sum_{\substack{\forall j \\ \wedge\tau_j\in\mathcal{T}^{\mathrm{nm}}_{P_k}}} \Big(N^{\mathrm{G}}_{S_{P_k},j} \times \max_{\forall R_q\in\mathcal{RS}^{\mathrm{G}}_j}\{Cs_{j,q}\}\Big), \tag{14}$$

where $N^{\mathrm{G}}_{S_{P_k},j}$ is the maximum number of such pi-blocking that S_{P_k} may experience from a lower priority task τ_j on P_k and is upper bounded as follows:

$$N^{\mathrm{G}}_{S_{P_k},j} = (\left\lceil \tfrac{T_s}{T_j}\right\rceil + 1)\times n^{\mathrm{G}}_j. \tag{15}$$

Proof: (14) is similar to (7). However, the blocking that is incurred to a task τ_i by lower priority tasks global resource access is due to the fact that τ_i gives opportunity to lower priority tasks (on the same core) to lock global resources when τ_i is blocked on a global resource and is suspended, and once before the task arrives. However, a (deferrable) server may leave the processor for more reasons. A server, may release the processor whenever there is no pending workload to be processed, or when the capacity of the server is depleted. Thus, server gives such opportunity to (lower priority) non-migrating tasks where they can contribute in delaying the server by preempting the server when they are granted access to the global resource (Rule 4). However, $N^{\mathrm{G}}_{S_{P_k},j}$ is still upper bounded by the maximum number of times that each (lower priority) non-migrating task can request global resources. This leads the upper bound of $N^{\mathrm{G}}_{S_{P_k},j}$ to be the second term of the minimum function in (8). ■

Total Server Delay. Followed by Lemmas 1 and 2, the maximum incurred delay to a server S_{P_k} on a processor P_k which has the highest priority compared to any non-migrating task on P_k is calculated as follows.

$$\delta_{S_{P_k}} = B^{\mathrm{G}}_{S_{P_k}}. \tag{16}$$

B. Global Synchronization Approach

In this section we provide the blocking analysis under our proposed protocol for migrating tasks which share resources by other migrating tasks scheduled within the same set of servers. Later in Section VI we show how these blocking terms are incorporated in the response time analysis of migrating tasks. A migrating task that is scheduled globally within a set of servers, may experience two types of blocking due to other migrating tasks requesting resources: (i) direct blocking on a resource that is hold by another migrating task, and (ii) blocking incurred by a lower priority migrating task when its priority is boosted due to a resource request (Rule 5 which holds also for migrating tasks). To calculate the maximum incurred blocking to a task due to case (i) and case (ii), we present Lemmas 3 and 4, respectively.

Lemma 3. *The maximum amount of time that a task $\tau_i \in \mathcal{T}^{\mathrm{m}}$ has to wait to lock a global resource R_q is denoted by $DB_{i,q}$ (direct blocking) and is calculated as follows:*

$$DB_i = \sum_{\substack{\forall q: R_q \in \mathcal{RS}^{\mathrm{G}}_i \\ \wedge \tau_i \in \mathcal{T}^{\mathrm{m}}}} (n^{\mathrm{G}}_{i,q} \times DB_{i,q}), \tag{17}$$

where,

$$DB_{i,q} = \sum_{\substack{\forall j: \tau_j \neq \tau_i \wedge \tau_i, \tau_j \in \mathcal{T}^{\mathrm{m}} \\ \wedge R_q \in (\mathcal{RS}_j \cap \mathcal{RS}_i)}} Cs_{j,q}. \tag{18}$$

Proof: Under the worst-case scenario, every time that τ_i requests a resource R_q, all other migrating tasks that use the same resource have requested it earlier and are placed ahead of τ_i in R_q's related FIFO queue. The worst-case scenario under which all tasks that request R_q may get blocked is imaginable where one task in the server is using R_q while these requests are issued one by one by tasks on other servers located on other core(s) and are not satisfied. Therefore all the tasks that have been placed ahead of τ_i in the resource FIFO queue will access the resource and delay τ_i. This scenario may happen every time τ_i requests a resource. As a result, τ_i is delayed, in the worst-case, for all its resource requests, by critical sections of all other tasks that have requested the same resource. This maximum delay is formulated in (17). ■

In Section VI we show how this blocking term is incorporated in the response time of a task τ_i based on the analysis described in Section IV-A.

Lemma 4. *Pi-blocking that is incurred to a task $\tau_i \in \mathcal{T}^{\mathrm{m}}$ due to non-preemptable execution of lower priority migrating tasks is denoted by NPB_i and is upper bounded as follows:*

$$NPB_i = \sum_{\substack{\forall j,q: \rho_j < \rho_i, R_q \in \mathcal{RS}_j \\ \wedge R_q \notin \mathcal{RS}_i \wedge \tau_i, \tau_j \in \mathcal{T}^{\mathrm{m}}}} \Big((\left\lceil \frac{T_i}{T_j} \right\rceil + 1) \times n^{\mathrm{G}}_{j,q} \times Cs_{j,q} \Big). \tag{19}$$

Proof: This term is in fact the maximum amount of time that tasks with priority lower than that of τ_i execute critical sections non-preemptively, (Rule 5) and interfere with τ_i's execution. Let us assume that τ_i is executing within a server S_{P_k}. Under a worst-case scenario, as soon as migrating tasks with priority lower than that of τ_i which are executing in servers on cores other than P_k, are granted access to a resource, the servers on those cores may get preempted (by non-migrating tasks that are granted access to a global resource). Remember that, servers are fully preemptive by resource access of non-migrating tasks (see Rule 5). As a result, those tasks with lower priority than τ_i will preempt τ_i in S_{P_k} and will execute their critical sections on that only available server (assuming that S_{P_k} on this core has not been preempted). In the worst-case, all tasks with priority lower than τ_i may arrive and run in the servers on cores other than P_k while τ_i is executing in S_{P_k} and in the same way contribute in delaying τ_i when they are granted access to a resource. This scenario may happen for every resource access of tasks with lower priority than τ_i. Therefore, the maximum amount of such delay that τ_i may experience is the summation of all critical sections of tasks with lower priority than τ_i. For a lower priority task τ_j, this amount for a specific resource R_q, is calculated by the maximum release time of τ_j during τ_i's period and accounting for τ_j's maximum number of requests on R_q. This term for a lower priority task τ_j is calculated for all τ_j's requests. Note that, the critical sections that are also used by τ_i are not considered in this term, since they are considered in DB_i term (Lemma 3). ■

VI. Response Time Analysis

We denote the worst-case response time of a non-migrating task $\tau_i \in \mathcal{T}^{\mathrm{nm}}_{P_k}$ by WR^{nm}_i which is calculated based on the classical response time analysis [14]. τ_i is schedulable if $WR^{\mathrm{nm}}_i \leq D_i$. The execution of non-migrating tasks with higher priority than that of a task $\tau_i \in \mathcal{T}^{\mathrm{nm}}$ can be delayed due to waiting for a global resource which can increase the interference to τ_i. This is addressed by incorporating RWT_j, which is the delay incurred to a non-migrating higher priority task τ_j due to resource waiting time, in (20) similar as done in [5], [13]. WR^{nm}_i is the smallest solution to the equation below.

$$t = C_i + B_i + \sum_{\substack{\forall \rho_j > \rho_i \\ \tau_i, \tau_j \in \mathcal{T}^{\text{nm}}_{P_k}}} \lceil (t + RWT_j)/T_j \rceil C_j \\ + (\lceil (t - C_{P_k})/T_s \rceil + 1) C_{P_k}. \tag{20}$$

Note that, RWT_j and B_i are calculated according to (12) and (13), respectively. Further, the last term in (20) accounts the interference of the server on the same core as τ_i (if any) [14]. In [19] a response time analysis has been presented for tasks scheduling under a fixed priority global scheduling on a multiprocessor platform. In this analysis tasks have full access to the processor bandwidth. However, since in our system model tasks that are globally scheduled have partial access to the processor bandwidth on a core (by means of servers), we shall use the response time analysis that has been developed in [31] (see Section IV-A) which is suitable for such a system model. Next, we extend the response time analysis in [31] to account for resource sharing parameters. Since on each core, a server can be viewed as a task which share no resources with the non-migrating tasks on that core, scheduling the server along with non-migrating tasks on the core will lead to the server experiences a delay from those tasks accessing resources. This maximum delay to the server on a core has been presented by (16). Moreover, due to resource sharing among migrating tasks, a task $\tau_i \in \mathcal{T}^{\text{m}}$ may experience an extra delay when it is processed within servers which is presented by (17) and (19). As a result, in the worst-case, a migrating task τ_i experiences the same delay that is incurred to the server as well as the delay that is caused due to resource sharing by migrating tasks. In the following we show how these two different types of delays are incorporated in the response time of a task $\tau_i \in \mathcal{T}^{\text{m}}$.

For a migrating task that share resources in the system, we introduce a new concept of head when resource sharing is enabled in the system, which is more general compared to the notion used in [31]. We call this new concept of head, the *starvation period*, which is depicted in Figure 1. We refer to starvation period as the interval that the job cannot have access to any server capacity, which is constructed by two intervals: (i) the interval between the activation of the job (arrival time of the job) and the first server replenishment (which is similar to the previous notion of head in [31] and we still call it head) and (ii) the interval in which the task looses the server capacity due to waiting for its resource requests, which we refer to as *capacity loss interval*. The body is the rest of the scheduling window interval, i.e., until the job finishes. It is obvious that if the task inside the server does not use any resource, the starvation period is similar to the previous notion of head in [31]. The starvation period for a task $\tau_i \in \mathcal{T}^{\text{m}}$ is denoted as follows:

$$\widehat{Sv}_i^{\text{C}} = \widehat{H}_i^{\text{C}} + \widehat{Cap}_i^{\text{loss}}. \tag{21}$$

The capacity loss interval for a migrating task τ_i is the time needed for the maximum execution workload of critical sections related to other migrating tasks that request the same resources as τ_i. This workload is in fact the DB_i term described in Lemma 3. Under the worst-case assumption, this workload is executed in serial on one processor (since in the worst-case all servers except one may get preempted by non-migrating tasks on their cores, hence this workload is executed on the only available server). Moreover, under the worst-case assumptions, the only available server is the server with minimum capacity, i.e., $C_s^{\min}$. Followed by the above discussion, the maximum interval of capacity loss that τ_i may experience due to its resource requests is denoted by $\widehat{Cap}_i^{loss}$ and is upper bounded as follows:

$$\widehat{Cap}_i^{\text{loss}} = \left\lceil \frac{DB_i}{C_s^{\min}} \right\rceil \times T_s. \tag{22}$$

This interval has been shown in Figure 1 for a task τ_i. To simplify the illustration of the capacity loss interval concept, in this figure, we show an example for a task τ_i with one resource request. However, the idea of the capacity loss interval is to account for all τ_i's resource requests.

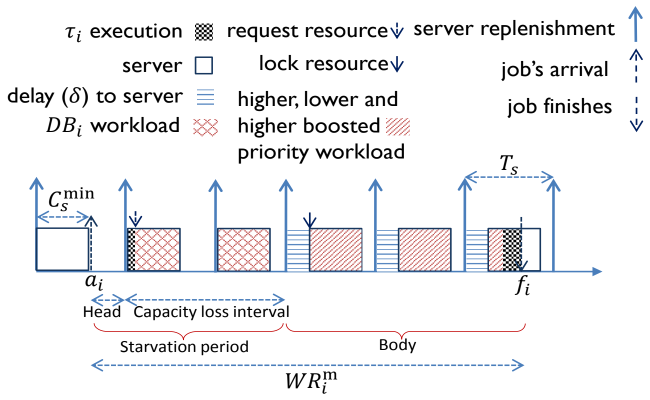

Fig. 1. migrating task τ_i's scheduling window.

As denoted in Section IV-A, the body constitutes of the time needed to process the workload of tasks with higher, lower priority and the task itself. Since for a task τ_i, NPB_i is the workload of tasks with higher boosted priority, it is treated similar to the workload of the higher priority tasks. As a result the body for task τ_i when resource sharing is enabled is updated as follows:

$$\widehat{B}_i^{\text{C}}(t) = \widehat{B}_i^{\text{C}}(t) + NPB_i. \tag{23}$$

As discussed before (in the early part of this section and also Section V-A), a server, regardless of tasks executing inside it, can be blocked by the non-migrating tasks on the same core. This means that a task τ_i 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 capacity and in the worst-case the delay caused by these tasks will only defer the server execution. Therefore, it is enough to add such delay once to the total response time of τ_i. The delay incurred to the server on a processor P_k has been presented in (16). Since a migrating task τ_i is scheduled globally, it may be scheduled on any of the servers. As a result, the maximum delay which τ_i may experience due to the delay incurred to a server, is calculated by finding the largest delay imposed to any server. This is presented by the last term in (24). As a result, the response time of a migrating task τ_i denoted as WR_i^{m} (2) is updated as follows, where, WR_i^{m} is the smallest solution to (24).

$$t \leq \widehat{Sv_i^{\mathrm{C}}} + \widehat{\acute{B}_i^{\mathrm{C}}}(t) + \max_{\forall P_k \wedge \tau_i \in \mathcal{T}^{\mathrm{m}}} \delta_{S_{P_k}}. \tag{24}$$

Similar to the analysis of non-migrating tasks (in (20)), the effect of resource waiting times of other tasks needs also to be taken into account in the response time of migrating tasks. The number of interference that a migrating task τ_j may incur to another migrating task τ_i, may increase due to resource waiting time of τ_j. It has been shown in [31] (Lemma 4.2) that the response time of a task scheduled within a set of servers, is upper bounded when execution of the task starts after all the workload of tasks with higher and lower priority is completely finished. This means that both tasks with higher and lower priority incur interference to the execution of a task processed within a server. Therefore, resource waiting times of both tasks with higher and lower priority needs to be considered in the response time of a task τ_i. In the following we show how this delay is considered in the response time of a task τ_i.

$RW_j^i(L)$, as presented in [31], is denoted as the upper bound of the requested workload of a task τ_j ($\tau_j \neq \tau_i$) in the interval L, where L is the interval of τ_i's response time (see Appendix C). This term is part of the body of a task τ_i, i.e., it has been included in $\widehat{\acute{B}_i^{\mathrm{C}}}(t)$ in (24) ([31], see also (Appendix C)). In order to account for extra released workload of a migrating task τ_j (with priority either higher or lower than τ_i) due to its resource waiting time when we calculate the response time of τ_i, L should be extended to include those resource waiting times . Further, in (22) we have shown that how the resource waiting time of a task may be further extended to several server periods in the worst-case. Therefore, to account for the worst-case, L is extended by $\widehat{Cap_j^{\mathrm{loss}}}$, i.e., $L = L + \widehat{Cap_j^{\mathrm{loss}}}$ as presented in (25).

Moreover, the execution related to critical sections of tasks with priority higher and lower than a task τ_i has been once considered in the response time of τ_i by the blocking terms $DB_{i,q} : \forall R_q \in \mathcal{RS}_i$ and NPB_i in (17) and (19), respectively (see (22), (23)). Therefore, to make the analysis tighter, we need to remove these executions from the worst-case execution time of those tasks when we calculate the higher and lower priority workload presented in (25). Therefore, we remove the execution of critical sections that is considered in (17) and (19)from the execution time of any task τ_j that interfere with execution of τ_i. Thus, the execution time of τ_j is updated in (25) according to (26).

$$\begin{aligned} &\forall j \neq i : RW_j^i(L) = N_j(L) \times \acute{C}_j + \\ &min(\acute{C}_j, L + \widehat{Cap_j^{\mathrm{loss}}} + D_j - \acute{C}_j - N_j(L) \times T_j), \end{aligned} \tag{25}$$

where $N_j(L) = \left\lfloor \frac{L + \widehat{Cap_j^{\mathrm{loss}}} + D_j - \acute{C}_j}{T_j} \right\rfloor$, and

$$\acute{C}_j = \begin{cases} C_j - \sum\limits_{\forall q: R_q \in (\mathcal{RS}_j \cap \mathcal{RS}_i)} n_{i,q}^{\mathrm{G}} \times Cs_{j,q} & \text{if } \rho_j > \rho_i \\ C_j - \sum\limits_{\forall q: R_q \in (\mathcal{RS}_j \cap \mathcal{RS}_i)} n_{i,q}^{\mathrm{G}} \times Cs_{j,q} & \\ - \sum\limits_{\substack{\forall p: R_p \in \mathcal{RS}_j \\ \wedge R_p \notin \mathcal{RS}_i}} (\left\lceil \frac{T_i}{T_j} \right\rceil + 1) \times n_{i,q}^{\mathrm{G}} \times Cs_{j,p} & \text{if } \rho_j < \rho_i \end{cases} \tag{26}$$

VII. Evaluation

In this section we present the results of experiments, illustrating the effect of system parameters, such as server period, on the schedulability of a system with resource sharing. In our experiments we make sure that the partitioned tasks are schedulable, i.e., if it is not schedulable, we discard the whole platform. The non-migrating tasks being schedulable, we investigate the schedulability of the migrating tasks. In our experiments, we have investigated two different heuristic algorithms which results in balanced and unbalanced allocation of non-migrating tasks on the platform. In the first approach which we denote as B in the graphs, we use the worst-fit allocation technique. Based on the worst-fit algorithm, for assigning a task, the processor with the minimum utilization of assigned tasks is chosen. In the second approach we use the first-fit allocation technique. Under the first-fit algorithm, the first processor from the processor list that can fit the task is chosen for a task to be assigned to. After non-migrating tasks are allocated to the cores, the slack on each core is evaluated and is dedicated as the budget of the server allocated to that core. The algorithm of finding the slack on each core is similar to [23], where we have incorporated the blocking terms in the demand bound function. This algorithm can be seen in the Appendix B. We have investigated the use of the first-fit algorithm under two cases: (1) we use first-fit and allocate servers to all cores that can provide capacity for server, this approach is denoted as $UnB1$ in the graphs, (2) we don't allocate server to the processor that provides a small capacity, this approach is denoted as $UnB2$ in the graphs. In the second case, if the server with the minimum capacity provides a budget less than 15% of the server period, the server is discarded, i.e., we don't allocate any server to the related core.

A. Experimental Setup

In each experiment 10000 task sets are randomly generated. In the experiments we have checked the schedulability versus server replenishment period which varies in the range: [500, 4100] with granularity 400. In each experiment, the platform configuration selects: the number of processors m, non-migrating task set cardinality denoted by n^{nm}, migrating task set cardinality denoted by n^{m}, number of resources used per task $ResNum$ and resource usage rate $ResUsage$, i.e. the percentage of the tasks in the task set that use resources. $ResNum$ and $ResUsage$ are selected similar for both migrating and non-migrating tasks. The maximum length of critical sections is randomly selected from the range [50, 300] in each experiment. Number of resources used for both non-migrating and migrating tasks is fixed to 10.

The results presented in this section are for: $n^{\mathrm{nm}} = 10$ and $n^{\mathrm{m}} = 5$, number of processors $m = 4$, number of resources requested per task $ResNum = 2$, resource usage rate $ResUsage = 20\%$ and maximum critical section length of 65. More results for task set cardinality of 5, 10, 15, $m = 4, 8$, $ResNum = 2, 3, 5$ and $ResUsage = 20\%, 30\%, 50\%$ can be seen in the longer version of this paper [2].

We have used the UUnifast-Discard algorithm [18] for assigning tasks utilization of non-migrating tasks where the total task set utilization is set to $m/2$. For the first-fit allocation we have used a 0.7 per processor utilization cap which is a processor capacity criterion to fill a processor up to this limit. The inter-arrival time of the tasks of non-migrating tasks in each experiment is selected randomly from the range [1000, 10000]. Further, the per task utilization of migrating tasks is randomly selected from the range [0.01, 0.4] and the tasks inter-arrival time is selected from the range [5000, 20000] in each experiment. In the experiments, deadline of the tasks equals to their inter-arrival time.

B. Results

As can be seen in Figure 2, in general, by increasing the period of the server, the system schedulability decreases. This is due to the fact that, in general, by increasing the server period, the starvation period of a task which is scheduled inside the server increases (see (22) and (21)) which leads to an increase in the response time of the task. However, increasing the server period may also lead to an increase in the schedulability as can be observed in Figure 2 ($T_s = 2900$ in graph B). The reason is that increasing the server period may lead to an increase in the server budget on a core (see Appendix B, Algorithm 1, line 12). This in turn results in the total server capacity increase as also illustrated in Figure 3. On the other hand, when the total server capacity increases, more workload can be processed during a server period, which can lead to a decrease in the response time of the tasks, thus an increase in the schedulability. Such an effect is explained in more detail by an example shown in Figure 5 at the end of this section.

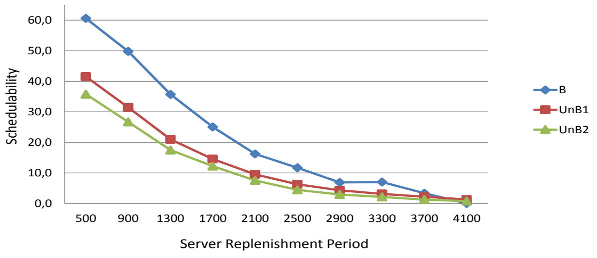

Fig. 2. Schedulability versus server replenishment period. $ResUsage$ = 30%, n^{nm} = 10, n^{m} = 5, m = 4, $ResNum$ = 2.

The experiments show a better schedulability result for when non-migrating tasks are partitioned unbalanced on the cores of the platform (Figure 2). The reason is that the total server capacity is larger under the unbalanced approach compared to the balanced approach. The reason for the larger total server capacity under the unbalanced approach is that by using the first-fit allocation technique, usually no (non-migrating) task is allocated to at least one processor since the algorithm tries to fill the first processors as much as possible. This means that the whole processor bandwidth is allocated to the server under this approach. The experiments show that in general, $UnB2$ has a higher schedulability compared to $UnB1$. The reason is as follows. Under the first-fit algorithm the minimum server capacity is usually smaller compared to the balanced approach sine the algorithm tries to fill the first cores as much as possible. The minimum server capacity influences the starvation period (21) by both affecting the head (3) and capacity loss intervals (22). A smaller minimum server capacity results in a larger starvation period and hence a larger response time. Therefore, by not allocating server to the core with the smallest capacity (if the smallest capacity has been smaller than a predefined range) under $UnB2$, we could remove the effect of the small minimum server capacity.

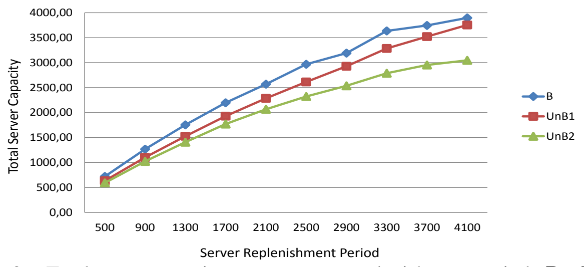

Fig. 3. Total server capacity versus server replenishment period. $ResUsage$ = 30%, n^{nm} = 10, n^{m} = 5, m = 4, $ResNum$ = 2.

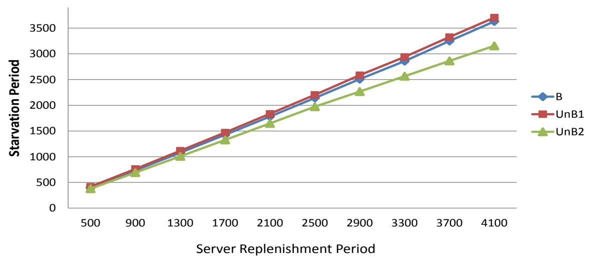

Fig. 4. Starvation period vs. server replenishment period. $ResUsage$ = 30%, n^{nm} = 10, n^{m} = 5, m = 4, $ResNum$ = 2.

Moreover, further results (in the longer version of this paper [2]) show that increasing the number of resources used per task, the task set's resource usage rate and task set size lead to a schedulability decrease which is not a surprising result since the mentioned factors cause an increase in the blocking delay incurred by tasks. Increasing the number of migrating tasks, causes the workload of higher and lower priority tasks that should be processed before a task τ_i increases which results in increasing the body and consequently response time of the task (23), (24).

The results show that in general, by increasing the server period, the schedulability decreases. However, sometimes this trend does not hold for some points (in Figure 2 at period 2900 under the B approach). As it can be seen in Figure 3, the total server capacity increases sharply at this period, resulting in an increase in the schedulability at this point. This behavior is also related to task scheduling inside a server. To explain this effect we use a simple example shown in Figure 5. In this example we test the schedulability of a task inside the server when selecting three different periods for the server. The server periods are denoted as T_1, T_2 and T_3, where $T_1 < T_2 < T_3$. The resulted total server capacities are denoted as Q_1, Q_2 and Q_3 for each server period T_1, T_2 and T_3, respectively. Since by increasing the server period the total server capacity increases therefore: $Q_1 < Q_2 < Q_3$. Consider a task τ_i with execution time C_i where $C_i = Q_2 + \epsilon < 2Q_1 \wedge C_i \leq Q_3$. Assuming the worst-case release scenario for τ_i, as shown in Figure 5 denoted as a_i and the absolute deadline of the task denoted as d_i, by the assumption $C_i < 2Q_1$ the task can meet its deadline in case (a) since two server capacities are provided. However, since $C_i = Q_2 + \epsilon$, τ_i will miss its deadline in case (b) since only one server capacity is provided. Finally, in case (c) the task is again schedulable since $C_i \leq Q_3$ and at least one server capacity is provided. This implies that by increasing the period of the server, a task τ_i can change its status from being schedulable to non-schedulable and vice versa. This behavior is not related to the resource sharing parameters and the same trend has also been seen in [31], [30] where no resource sharing was provided.

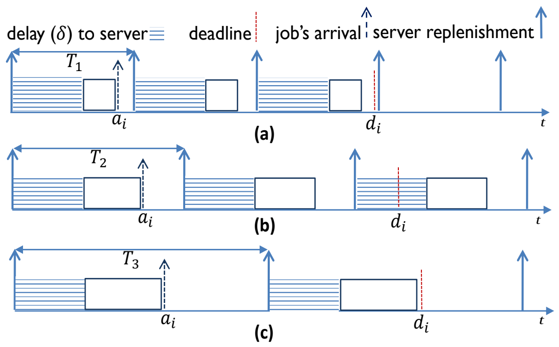

Fig. 5. Schedulability behavior vs. server period.

VIII. Conclusion and Future Work

In this paper, we enable resource sharing for platforms with critical and non-critical applications where each application has its own dedicated set of resources. Critical application is partitioned over the platform and the additional non-critical application is scheduled globally within a set of servers by using the remaining capacity on each core. We provide resource handling with the tailored blocking analysis for tasks that are scheduled globally by means of a set of servers where tasks have partial access to processors bandwidths. We present the new response time analysis which takes into account the provided blocking bounds. Further, we perform experimental evaluations where we investigate the system schedulability under balanced and unbalanced allocation of tasks of the critical application(s) to cores. The results show that the unbalanced approach has better schedulability compared to the balanced approach. As future work we plan to enable inter-application resource sharing for such a platform and investigate other resource sharing alternatives such as a spin-based approach.

References

[1] AUTOSAR release 4.0. 2012, http://www.autosar.org.
[2] S. Afshar, M. Behnam, R. Bril, and T. Nolte. On resource sharing under global scheduling with partial processor bandwidth. In *Mälardalen University, Tech. Rep., 2015 [Online]. Available: http://www.es.mdh.se/publications?author=361*.
[3] S. Afshar, N. M. Khalilzad, F. Nemati, and T. Nolte. Resource sharing among prioritized real-time applications on multiprocessors. In 6th *International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*, Dec. 2013.
[4] J. H. Anderson, S. K. Baruah, and B. B. Brandenburg. Multicore operating-system support for mixed criticality, 2009.
[5] N. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline-monotonic approach. In *Proc. IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, 1991.
[6] T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, 1991.
[7] T. Baker. A comparison of global and partitioned EDF schedulability tests for multiprocessors. Technical report, In International Conference on Real-Time and Network Systems, 2005.
[8] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In 28th *IEEE International Real-Time Systems Symposium (RTSS)*, pages 149–160, 2007.
[9] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In 13th *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 14th *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 31st *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 9th *IEEE/ACM Intl. Conference on Embedded Software (EMSOFT)*, pages 69–78, Oct. 2011.
[13] B. B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.
[14] G. C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.
[15] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In 19th *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 247–258, Jul. 2007.
[16] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *HANDBOOK ON SCHEDULING ALGORITHMS, METHODS, AND MODELS*. Chapman Hall/CRC, Boca, 2004.
[17] R. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In 26th *IEEE International Real-Time Systems Symposium (RTSS)*, pages 389–398, Dec. 2005.
[18] R. Davis and A. Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In 30th *IEEE Real-Time Systems Symposium (RTSS)*, pages 398–409, 2009.
[19] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In 30th *IEEE Real-Time Systems Symposium (RTSS)*, pages 377–386, Dec. 2009.
[20] 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 9th *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 189–198, May 2003.
[21] G. Lipari and E. Bini. A framework for hierarchical scheduling on multiprocessors: From application requirements to run-time allocation. In 31st *IEEE Real-Time Systems Symposium (RTSS)*, Nov 2010.

[22] G. Lipari and G. Buttazzo. Resource reservation for mixed criticality systems. In 14th *Workshop on Real-Time Systems: the past, the present, and the future, York, UK*, 2013.

[23] M. Liu, M. Behnam, S. Kato, and T. Nolte. A server-based approach for overrun management in multi-core real-time systems. In *The 19th IEEE International Conference on Emerging Technologies and Factory Automation*, September 2014.

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

[25] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. In 17th *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 269–279, April 2011.

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

[27] 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.

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

[29] A. Wieder and B. Brandenburg. On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks. In 34th *IEEE Real-Time Systems Symposium (RTSS)*, pages 45–56, Dec. 2013.

[30] 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.*

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

Appendix A
Notations

Here are the most used notations 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.
f_i: finishing time 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}: server dedicated to core P_k.
C_{P_k}: capacity of S_{P_k}.
$C_s^{\min}$: minimum capacity among servers.
ρ_{P_k}: priority of S_{P_k}.
T_s: server replenishment period.
$\mathcal{T}^{\mathrm{m}}$: set of migrating tasks.
R_q: resource q.
$\mathcal{R}_{P_k}^{\mathrm{L}}$: set of local resources accessed by tasks on P_k.
$\mathcal{RS}_i^{\mathrm{L}}$: set of local resources accessed by jobs of τ_i.
$\mathcal{RS}_i^{\mathrm{G}}$: 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_i^{G}: maximum number of τ_i's global requests.
$n_{i,q}^{\mathrm{G}}$: maximum number of requests of τ_i for global resource R_q.
$\mathcal{RS}_i$: set of resources accessed by jobs of τ_i.
$n_{i,q}$: number of τ_i's requests on R_q.
WR_i^{nm}: worst-case response time of a non-migrating task τ_i.
WR_i^{m}: worst-case response time of a migrating task τ_i.

Appendix B
Processor Slack

In this section, we describe how to assign server budget based on the slack on a processor. In order to find the slack on a processor P_k, the minimum slack among tasks with priority lower than that of the server (for simplicity we assume only one application per core) is found and not among the higher priority tasks. This is due to the fact that the 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 $\lceil T_j/T_s \rceil + 1$ to assign the budget for one server period. Note that the plus 1 is to account for the back-to-back execution anomaly [26] which happens in case of self-suspensions which is applied for a deferrable server. The final assigned server budget on processor P_k ,i.e., C_s, is determined in line 14 by finding the smallest calculated budget value.

Algorithm 1 Processor P_k Budget Assigning Algorithm

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

To find the slack of a task $\tau_j \in \mathcal{T}_{\mathrm{P_k}}^{\mathrm{nm}}$, the difference between the incurred load to τ_j 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_j's period (lines 10 to 19 in Algorithm 2). The task τ_j'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).

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 $\tau_i \in \mathcal{T}^{\mathrm{m}}$ is calculated for $t = WR_i^{\mathrm{m}}$ and is as follows:

$$R_{\mathrm{HL/i}}(t) = \left(\left\lceil \frac{W_{\mathrm{HL}}^i(t)}{\sum_{k=1}^{m} C_{P_k}} \right\rceil - 1\right).T_s + t_{\mathrm{res}}^{\mathrm{HL}}(t), \tag{27}$$

Algorithm 2 FindSlack(τ_j)

```
1: Initialize $checkPoint \leftarrow 0$
2: Initialize $checkPointList \leftarrow \oslash$
3: Initialize $hpTaskList \leftarrow \oslash$
4: $\tau_i \in \mathcal{T}^{\text{nm}}_{\text{P}_\text{k}}$
5: for all $\tau_h \in \mathcal{T}^{\text{nm}}_{\text{P}_\text{k}}$ do
6:     if $\rho_h > \rho_i$ then
7:         $\tau_h$ added to $hpTaskList$
8:     end if
9: end for
10: for all $\tau_h \in hpTaskList$ do
11:     k $\leftarrow 1$
12:     $checkPoint$ = $k \times T_h$
13:     while $checkPoint < D_i$ do
14:         if $checkPoint \notin checkPointList$ then
15:             $checkPoint$ added to $checkPointList$
16:         end if
17:         $k++$
18:     end while
19: end for
20: $D_i$ added to $checkPointList$
21: $load \leftarrow 0$
22: $maxTaskSlack \leftarrow 0$
23: for all $t \in checkPointList$ do
24:     $hpInterference \leftarrow 0$
25:     $slack \leftarrow 0$
26:     for all $\tau_h \in hpTaskList$ do
27:         $hpInterference += \left\lceil \frac{t}{T_h} \right\rceil \times C_h$
28:     end for
29:     $load = C_i + hpInterference + B_i$
30:     $slack = t - load$
31:     if $slack > maxTaskSlack$ then
32:         $maxTaskSlack \leftarrow slack$
33:     end if
34: end for
35: if maxTaskSlack $< 0$ then
36:     return $-1$
37: end if
38: return $maxTaskSlack$
```

where, $W^i_{HL}(WR^{\text{m}}_i)$ is specified according to (33) and t^{HL}_{res} is calculated as below:

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

$$W^{\text{HL}}_{\text{res}}(t) = W^i_{\text{HL}}(t) - (\lceil \frac{W^i_{\text{HL}}(t)}{\sum_{k=1}^{m} C_{P_k}} \rceil - 1).\sum_{k=1}^{m} C_{P_k}. \quad (28)$$

$$\forall k : 1 \leq k \leq m : \delta(k) = \sum_{p=k}^{m} C_{P_p} + C_{P_k}.(k-1). \quad (29)$$

$R_{i/HL}$ 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:

$$R_{i/\text{HL}} = \begin{cases} C_i & \text{if } C^{\text{rmn,i}}_s \geq C_i \\ T_s - t^{\text{HL}}_{\text{res}}(t) + CRP_i.T_s + C_i - & \\ CRP_i(t).min(\sum_{k=1}^{m} C_{P_k}, T_s) & \text{otherwise} \end{cases} \quad (30)$$

where,

$$CRP_i(t) = \lceil \frac{C_i - C_{\text{rmn}}(t)}{\sum_{k=1}^{m} C_{P_k}} \rceil - 1, \quad (31)$$

$$C_{\text{rmn}}(t) = min(\sum_{k=1}^{m} C_{P_k} - W^{\text{HL}}_{\text{res}}(t), T_s - t^{\text{HL}}_{\text{res}})(t). \quad (32)$$

$$W^i_{\text{HL}}(t) = W^i_{\text{HP}}(t) + W^i_{\text{LP}}(t), \quad (33)$$

where $W^i_{HP}(WR^{\text{m}}_i)$ and $W^i_{LP}(WR^{\text{m}}_i)$ 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. [31] showed that the workload of the tasks with lower priority than that of task τ_i are also affected by the response time of τ_i and is calculated as follows:

$$W^i_{\text{LP}}(t) = min(\sum_{j<i} RW^i_j(t), CCL_i(t)), \quad (34)$$

where for $t = WR^{\text{m}}_i$, $CCL_i(t)$ can be bounded from above as $CCL_i(WR^{\text{m}}_i) = (m-1).C_i$ and $RW^k_j(t)$ is calculated according to (36).

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

$$W^i_{\text{HP}}(t) = \sum_{j>i} RW^i_j(t), \quad (35)$$

where $RW^i_j(WR^{\text{m}}_i)$ 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 (36). However, due to resource access waiting time of a higher priority task τ_j, more workload related to τ_j may be released and contribute in $RW^i_j(t)$. As a result, we update $RW^i_j(t)$ to account for such extra workload according to (25).

$$\forall j \neq i : \\ RW^i_j(t) = N_j(t).C_j + min(C_j, t + D_j - C_j - N_j(t).T_j), \quad (36)$$

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

Further, by assuming that a server S_{P_k} can have an arbitrary priority on its allocated core, it may experience interference from other higher priority servers or non-migrating tasks assigned to the same core. IS_{P_k}, $1 \leq k \leq m$, denotes the maximum interference which a server S_{P_k} could suffer on a core P_k which can be calculated according to uniprocessor scheduling [17]. Considering such interference to a server, the maximum response time of a migrating task processed by the server then is calculated as below [30]:

IS_{max} is is denoted as:

$$IS_{\text{max}} = \max_{1 \leq k \leq m}\{IS_{P_k}\} \quad (37)$$

Zhu et. al has shown in [30] that the maximum response time of a migrating task processed by the servers related to an application is bounded from above by its WR^{m}_i when each server has the highest priority, plus $2 \times IS_{max}$.