Integrating Independently Developed Real-Time Applications on a Shared Multi-Core Architecture*

Sara Afshar, Moris Behnam, Thomas Nolte
Mälardalen University, Västerås, Sweden
Email: {sara.afshar, moris.behnam, thomas.nolte}@mdh.se

ABSTRACT
The shift towards multi-core platforms has become inevitable from an industry perspective, therefore proper techniques are needed to deal with challenges related to this migration from single core architectures to a multi-core architecture. One of the main concerns for system developers in this context is the migration of legacy real-time systems to multi-core architectures. To address this concern and to simplify migration, independently-developed subsystems are abstracted with an interface, such that when working with multiple independently-developed subsystems to be integrated on a shared platform, one does not need to be aware of information or policies used in other subsystems in order to determine subsystem-level schedulability. Instead schedulability can be checked through their interfaces at the time of integration on a shared multi-core architecture. In this paper we propose a solution for investigating the system schedulability via providing interfaces for independently-developed subsystems where some of them are distributed over more than one processor and may share resources.

1. INTRODUCTION
Moving towards multi-core technology in industry has raised an increased interest to investigate real-time scheduling policies and system performance studies of multiprocessor subsystems in the real-time community. One of the main concerns while shifting to multi-core platforms is the *existing subsystems*. It is desirable that existing subsystems can co-execute on a shared platform without significant loss of performance.

A major challenge for integrating independently-developed subsystems, for example legacy systems, into a shared multi-core platform is how to integrate these subsystems with minor changes and how to abstract their resource demands comprehensively such that each subsystem is allowed to be unaware of the policies used in other subsystems.

Integrating multiple independently-developed subsystems on a shared multi-core platform, different alternatives may come up related to allocation of the subsystems to processors. One scenario is that each subsystem fits in one exclusive processor, i.e., no two subsystems share one core (processor), which has been studied in [1]. Another alternative is that one processor contains more than one subsystem. For this scenario, the techniques for integrating subsystems on uniprocessors can be used, e.g., the methods presented in [2] and [3]. These techniques abstract the timing requirements of the internal tasks of each subsystem which, as a result abstracts each subsystem as one (artificial) task. Therefore the problem of integration becomes similar to the case of scheduling a set of tasks running on a single processor. We can see that by reusing uniprocessor techniques for the second scenario it becomes similar to the first alternative.

The third alternative represents the scenario when a subsystem is allocated over more than one processor, which is also the focus of our paper. The challenge here is to provide predictable co-execution of the independently-developed subsystems, despite of how many processors each subsystem may be distributed over, considering that each subsystem may share resources.

In this paper we generalize the idea in [1] such that some subsystems, which we will call *applications* in the remainder of the paper, are allocated to more than one processor. The goal of the paper is to provide a solution which enables the schedulability analysis of integrated independently-developed applications which may be allocated over more than one processor without the application level scheduling knowledge. Targeting independently-developed applications allocated to more than one processor, we perform compositional schedulability analysis, i.e., we check schedulability of the system by composing interfaces that abstract schedulability requirements of each application [4]. Using compositional analysis, the system integrator can investigate if the whole platform is schedulable without any need to perform application level schedulability analysis. This is significant since (i) the application developer does not need to have detailed knowledge of scheduling policies or techniques used in other applications that are going to be integrated with this application on a shared platform, and (ii) to check the schedulability of the system, the system developer does not need to know detailed information on the task level of each application when integrating the applications.

In the context of multiprocessor scheduling there are two conventional scheduling techniques: partitioned and global scheduling. Under partitioned scheduling, each task is assigned to one processor and execute exclusively on that processor. On the other hand, under global scheduling tasks are allowed to migrate among processors and execute on any available processor. Semi-partitioned scheduling is a third alternative, introduced by Anderson et al. in [5] which extends partitioned scheduling by allowing a few tasks to migrate among different processors and improves the schedulability performance for independent task systems. Looking at the challenges related to the applications requiring more than one processor, we will look at semi-partitioned approach as an alternative for partitioning since it utilizes the resources in a better way as we will explain in Section 7. In this paper we investigate the partitioned and semi-partitioned approaches to partition applications

*This work is supported by the Swedish Foundation for Strategic Research, via the research program PRESS.

which do not fit on one processor, and we present techniques to abstract and derive interfaces for applications under these alternatives. The paper contributions are as follows:

- Targetting independently-developed applications that are allocated to more than one processor, we extract an interface for each application which abstracts the application resource demands.
- We propose the semi-partitioned approach as an alternative for partitioning the application on the processors/cores.
- We suggest the usage of multiple interfaces for different partitioning configurations, providing flexibility and better resource utilization.

The remainder of this paper is organized as follows: in Section 2 we present related work and in Section 3 we define our system model. We specify assumptions and rules of the synchronization protocol that manages sharing of resources in Sections 4 and 5 respectively. We perform subsystem analysis and abstract the timing requirement of each application in Section 6. Finally we investigate the subsystem abstraction by assuming partitioned and semi-partitioned approaches in Sections 7 and 8.

2. RELATED WORK

Vast amount of work has been done on the subject of integrating independently-developed real-time subsystems in a shared open environment on uniprocessors [3, 6, 7, 2]. Hierarchical scheduling techniques have been introduced and developed as a solution for these subsystems. Hierarchical scheduling has also been studied for multiprocessors [8, 9]. However, the subsystems studied in these works are assumed to be independent and they do not support sharing of mutually exclusive resources. In the context of resource management of non-hierarchical multiprocessor systems, a considerable amount of work has been done over the past decades.

Rajkumar et al. proposed the Distributed Priority Ceiling Protocol (DPCP) [10] for shared memory multiprocessors. In DPCP a job access its local resources and execute its non-critical sections on its assigned processor while it may access global resources on processors other than its assigned processor. The Multiprocessor Priority Ceiling Protocol (MPCP) was proposed by Rajkumar et al. [10, 11], which is an extension of the Priority Ceiling Protocol (PCP) [12] for multi-cores. In MPCP a task requesting a resource is suspended if the resource is not available at the moment. The Multiprocessor Stack Resource Policy (MSRP) is a resource sharing protocol proposed by Gai et al. [13], which extends the Stack Resource Policy (SRP) [14] for multiprocessors. Under MSRP the task that requests a global resource that is already locked by another task performs a busy wait denoted *spin lock*. The Flexible Multiprocessor Locking Protocol (FMLP) is a synchronization protocol introduced by Block et al. [15] for both partitioned and global scheduling, which later was extended to partitioned FMLP by Brandenburg and Anderson [16]. Under FMLP, resources are divided into long and short resources. Tasks that are blocked on long resources are suspended in the same way as MPCP while tasks that are blocked on short resources perform busy-wait similar to MSRP. The $O(m)$ Locking Protocol (OMLP) is another locking protocol proposed by Brandenburg and Anderson [17] to handle resource sharing in multiprocessors. However, the aforementioned synchronization protocols for multi-core/ multiprocessors do not support *compositional analysis* of independently developed applications. One of the semaphore-based synchronization protocols that supports integration of independently developed applications is the Multiprocessor Synchronization Protocol for Open Systems (MSOS) by Nemati et al. [1]. Under MSOS applications/ subsystems are developed independently and abstracted in their interfaces, therefore they do not need to have any knowledge about the scheduling algorithms and priority settings of other subsystems in order to determine schedulability. However, in [1] an application is assumed to be allocated to one core while in our work we relax this assumption and assume that an application can be distributed over multiple cores.

In the context of semi-partitioned scheduling, different allocation mechanisms have been investigated in prior works [18, 19, 20, 21], where Guan et al. have increased the utilization bound of task sets to achieve the utilization bound of Liu and Layland's Rate Monotonic Scheduling (RMS) for an arbitrary task set [21]. In these works, tasks are assumed to be independent, i.e., no resource sharing is allowed between tasks.

Inspired by our previous work on supporting resource sharing under semi-partitioned scheduling [22, 23], and based on the subsystem abstraction presented in [1], we propose a new approach to abstract independently-developed applications running on a multi-core platform where the applications are potentially requiring more than one core to be schedulable.

3. SYSTEM MODEL

In this section we present the system model used throughout this paper. We assume that the multi-core platform, which we call *platform* in the remainder of the paper, is composed of identical processors with shared memory. An application consists of a task set and a particular scheduling algorithm and tasks may request mutually exclusive resources. Some applications in the platform can fit on one processor while others do not and must be allocated over more than one processor. Note that applications do not share cores (processors), i.e., for each core only a complete or a part of an application can be allocated. The scheduling techniques for applications may differ between applications, e.g., one application may use a Fixed Priority Scheduling (FPS) policy, while another application may apply a dynamic priority scheduling policy (e.g., Earliest Deadline First EDF). However, due to space limitations and presentation clarity, in this paper we assume only the usage of FPS. A task set of an application A_k is denoted by τ_{A_k} and consists of n sporadic hard real-time tasks $\tau_i(T_i, C_i, D_i, \rho_i)$, where T_i identifies the minimum inter-arrival time between two successive jobs of task τ_i with worst-case execution time C_i and priority ρ_i. D_i represents the task's deadline where $D_i \leq T_i$. We also assume that each task in an application has a unique priority.

The tasks on application P_k share a set of resources R_{P_k} which are protected using semaphores. The set of shared resources R_{P_k} consists of two subsets of different types of resources; *local* and *global* resources. Local resources are shared by the tasks on the same processor while global resources are shared by tasks on more than one processor by the same or different applications. We denote the sets of local and global resources accessed by tasks on processor P_k as $R^L_{P_k}$ and $R^G_{P_k}$ respectively, i.e., $R_{P_k} = R^L_{P_k} \cup R^G_{P_k}$. We denote $C_{i,q}$ as the worst-case execution time of the longest critical section in which a task τ_i requests the resource R_q. Nested critical sections are not supported in this paper which in turn will remove the deadlock problem. However, tasks can access the same resource or more than one global resource sequentially.

According to the semi-partitioned approach some tasks are assigned to exactly one processor – we identify these tasks as non-split tasks. However, some tasks may be assigned to more than one processor within the same application. We refer to these tasks as split tasks since they are split among several processors. Each single part of a split task is called subtask. From an analysis point of view, all subtasks of each split task are assumed as normal separate tasks in the application, however, each subtask of a split task should execute prior to its successive subtask(s). We model this behavior using a constant offset, in the sense that each subtask of a split task has a constant offset according to its previous subtask.

We present each split task τ_s as a subset of l subtasks $(\tau_i^1, ..., \tau_i^l)$, and each subtask is represented by $(C_s^k, T_s, D_s, \rho_s, O_s^k)$ where $(k = 1, ..., l)$. O_i^k represent the constant offset of the k_{th} subtask of split task τ_s which is identified by the former subtask's maximum response time. The offset of the first subtask is zero, $O_i^1 = 0$. For the subtasks of a split task τ_s, T_s, D_s and ρ_s are the same as τ_s [23].

The resource requests of split tasks can happen at any time during the execution time of the task which means that it can happen in any core and not in a certain core. Therefore a conservative assumption from an analysis point of view is to assume that the critical sections of split tasks may happen in all subtasks of the split task and thus on different cores/processors. As the result, the resources requested by split tasks are by definition global resources [23].

4. DEFINITIONS AND ASSUMPTIONS

In order to perform the system-level schedulability analysis, we derive an interface for each application which reflects the scheduling demands of all its tasks. Note that, the tasks in each application do not need to be aware of any information about the tasks in other applications, neither do they need to know about scheduling and partitioning techniques used in other applications. We assume that each application A_i is allocated over a set of l processors, where $l \geq 1$ (the solution presented in [1] is only applicable for the case when $l = 1$). We denote the set of processors on which application A_i is allocated as P_{A_i}. We first specify some definitions and terms needed in this context.

4.1 Resource Hold Time

$\text{RHT}_{q,k,i}$ is the resource hold time and it defines the maximum time duration that the global resource R_q can be held by τ_i executing on P_k [1]. By definition, $\text{RHT}_{q,k,i}$ accounts for the longest critical section in which τ_i accesses R_q as well as the possible interference from other tasks accessing global resources other than R_q on processor P_k.

We also introduce two other terms as processor and application locking time for a specific global resource. Processor locking time of a processor P_k for a global resource R_q presented by $Z_{q,k}$ is the maximum duration of time that any task on processors other than P_k requesting R_q is blocked by tasks on P_k. In other words, $Z_{q,k}$ represents the impact of blocking on R_q introduced by P_k to all other processors. Furthermore, application locking time of an application A_i for a global resource R_q is denoted by Z_{q,A_i} and is the maximum duration of time that any task in applications other than A_i requesting R_q can be blocked by tasks in A_i.

4.2 Resource Wait Time

The maximum time duration that any task on processor P_k which requests global resource R_q may wait for the resource to be released and become available for the processor is identified as the resource wait time of processor P_k for global resource R_q and is presented by $\text{RWT}_{q,k}$. Similarly, the maximum duration of time that any task in an application A_i has to wait for a global resource R_q to be available is called resource wait time of application A_i for global resource R_q and is denoted by RWT_{q,A_i}.

4.3 Application Interface

An application A_i is abstracted by an interface $I_{A_i}(Q_{A_i}, Z_{A_i})$, where Q_{A_i} represents a set of requirements each extracted from a task in the application which requests global resources. If all requirements in the application are satisfied, then the application is implied to be schedulable. We target hard real-time applications, i.e., with all applications in the platform schedulable, then the platform becomes schedulable.

On the other hand, each application introduces different delays for different global resources to other applications in the platform that request those global resources. These delays are also abstracted in the interface of each application along with the requirements. Z_{A_i} in the interface of A_i represents these delays which is a set Z_{q,A_i} for each global resource R_q requested by A_i.

5. GENERAL DESCRIPTION

For each global resource in the system there is a unique FIFO queue in which the applications having tasks requesting the resource are enqueued. Note that, since the applications are independently developed, then the relative priority among tasks in different applications is not defined which makes the use of a FIFO queue preferable.

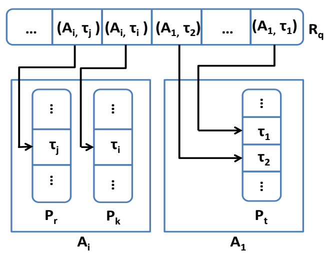

Figure 1: Resource queue management

Figure 1 shows a simple example of a system that consists of two applications running on 3 cores. At a certain time, τ_i is requesting a global resource and as the resource is not available the request is queued in the related resource global FIFO queue as shown in the picture. τ_i will be suspended since the request is not on the head of the global FIFO queue. P_r, P_k and P_t shows the processors each of which consists of a set of tasks assigned to them as illustrated in the picture.

5.1 Resource Sharing Rules

Rule 1: Local resource requests are handled by uniprocessor synchronization protocols such as PCP or SRP.

Rule 2: The priority of a task τ_i granting access to a global resource is immediately boosted to the value equal to $\rho_i + \rho^{max}(P_k)$, where $\rho^{max}(P_k) = \max\{\rho_i | \tau_i \in P_k\}$. It means that the task can preempt all higher priority tasks which do not use any global resource and all lower priority tasks even if they are accessing a global shared resource. This cause the blocking times of tasks to become a function of global critical sections (*gcs*) only.

Rule 3: When a task τ_i located on processor P_k related to an application A_i requests a global resource, τ_i access the resource if the resource is available (i.e., the global resource FIFO queue is empty) otherwise, a request for τ_i associated to A_i is added to the resource FIFO queue and τ_i is suspended.

Rule 4: When a global resource R_q becomes available to the application A_i the eligible task at the head of the processor waiting queue becomes ready to execute, and its priority is boosted.

Rule 5: When a task τ_i on processor P_k in application A_i releases a global resource R_q, then the placeholder of A_i will be removed from the resource queue and the resource becomes available to the processor whose application is at the top of R_q's queue. Also, the priority of the task will return to its normal value.

6. APPLICATION ANALYSIS

Here we elaborate the resource hold time and resource wait time of the tasks in the application level. We assume that applications use partitioned scheduling and in Section 7 we will extend the analysis for semi-partitioned scheduling.

In order for interfaces to enable the system schedulability analysis test we need to consider the worst case response time analysis for each task inside an application. Therefore we have to consider the maximum interference imposed to tasks due to resource sharing. The maximum time that an application A_i has to wait for R_q to be available for A_i occurs when all other applications in the system has requested R_q just before A_i and as a consequence are already waiting in the FIFO queue of R_q. Therefore, the resource wait time for A_i based on the interference from other applications Z_{q,A_j} is calculated as follows:

$$\mathrm{RWT}_{q,A_i} = \sum_{\forall A_j | A_j \neq A_i} Z_{q,A_j}. \quad (1)$$

According to Z_{q,A_i}'s definition along with the resource handling queue structure which is FIFO based, the maximum blocking time that A_i can introduce to any task τ_j in an application other than A_i occurs when all tasks in all processors of A_i that share R_q request R_q just before τ_j and their requests enqueued in the FIFO queue before τ_j's request. Note that, the longest time that R_q can be locked by processor P_k is $Z_{q,k}$. Therefore the application locking time of A_i on a global resource R_q is calculated as follows:

$$\mathrm{Z}_{q,A_i} = \sum_{P_k \in P_{A_i}} \mathrm{Z}_{q,k}. \quad (2)$$

As it can be seen, if A_i is allocated on one processor, then Z_{q,A_i} becomes similar to $Z_{q,k}$.

$Z_{q,k}$ is the maximum blocking time imposed by all tasks τ_q,k that share R_q and are located on P_k, on other tasks in other processors, e.g., τ_x. The maximum $Z_{q,k}$ happens when these tasks τ_q,k request R_q before τ_x. The maximum time that R_q can be locked by each element in τ_q,k is by definition $\mathrm{RHT}_{q,k,i}$. Thus the processor locking time of P_k on R_q is calculated as follows:

$$Z_{q,k} = \sum_{\tau_i \in \tau_{q,k}} \mathrm{RHT}_{q,k,i}. \quad (3)$$

On the other hand, the resource holding time of a global resource R_q accessed by τ_i based on the definition in Section 4.1 is computed as follows:

$$\mathrm{RHT}_{q,k,i} = Cs_{i,q} + H_{i,q,k}. \quad (4)$$

where $H_{i,q,k}$ denotes the interference from higher priority tasks, which is calculated as follows [1]:

$$H_{i,q,k} = \sum_{\substack{\rho_i < \rho_j \\ \wedge\, R_l \in R^G_{P_k},\, l \neq q}} Cs_{j,l}.$$

6.1 Blocking Terms

In this section we describe the possible scenarios where a task τ_i can be blocked by other tasks on the same or other processors.

6.1.1 Local blocking due to local resources

Each time a task τ_i is blocked on a global resource, it gives the chance to a lower priority task τ_j to lock a local resource, which in turn may block τ_i when it resumes after it releases the global resource. We represent the number of *gcs*'s of τ_i by n_i^G. The above mentioned scenario can happen up to n_i^G times. In addition, according to local synchronization protocols such as PCP and SRP, task τ_i can be blocked on a local resource by at most one critical section of a lower priority task which has arrived before τ_i. On the other hand, τ_j can release a maximum of $\lceil T_i/T_j \rceil$ jobs before τ_i's current job is finished. Furthermore, each job of τ_j can block τ_i's current job at most $n_j^L(\tau_i)$ times, where $n_j^L(\tau_i)$ denotes the number of critical sections in which τ_j requests local resources with ceiling higher than that of priority τ_i. Therefore, the blocking time on local resources, which is denoted by $B_{i,1}$, upper bounds as follows:

$$B_{i,1} = \min\{n_i^G + 1, \sum_{\rho_j < \rho_i} \lceil T_i/T_j \rceil n_j^L(\tau_i)\} \max_{\substack{\rho_j < \rho_i \\ \wedge\, R_l \in R^L_{P_k} \\ \wedge\, \rho_i \leq ceil(R_l)}} \{Cs_{j,l}\}, \quad (5)$$

where $ceil(R_l) = \max\{\rho_i | \ \tau_i \in \tau_{l,k}\}$.

6.1.2 Local blocking due to global resources

Each time τ_i suspends on a global resource, a lower priority task τ_j may access a global resource which subsequently can preempt τ_i after it resumes and finishes its *gcs* in its non-*gcs* sections. This situation may also happen when τ_j arrives sooner than τ_i. Therefore, this blocking can happen as many times as τ_i are requesting global resources up to n_i^G times in addition to the case where τ_j may arrive sooner than τ_i, which causes a maximum of $n_i^G + 1$ times.

Similar to the previous case, τ_j can release at most $\lceil T_i/T_j \rceil$ jobs before τ_i's current job finishes. On the other hand, each job of τ_j can preempt τ_i's current job a maximum of n_j^G times.

Thus, this kind of blocking introduced by τ_j to τ_i denoted by $B_{i,2}$ can happen at most $\min\{n_i^G + 1, \lceil T_i/T_j \rceil n_j^G\}$ times which is upper bounded as follows:

$$B_{i,2} = \sum_{\substack{\rho_j < \rho_i \\ \wedge\, \{\tau_i, \tau_j\} \subseteq \tau_{P_k}}} \Big(\min\{n_i^G + 1, \lceil T_i/T_j \rceil n_j^G\} \max_{R_q \in R^G_{P_k}} \{Cs_{j,q}\} \Big). \quad (6)$$

6.1.3 Remote blocking

When a task τ_i is blocked on a global resource which is already locked by a task on another processor, it is implied as remote blocking of task τ_i on that global resource. Based on our system design, when a task τ_i on processor P_k belonging to application A_i is blocked on a global resource R_q, it is added to the FIFO of R_q and it waits until it will be selected. To account for the maximum remote blocking that can be introduced to τ_i, we should assume that all applications have requested the same global resource that τ_i has requested before τ_i. At the same time, we should also assume that all tasks located on other cores within the same application as τ_i also requested the same global resource before the task. This scenario can happen each time τ_i requests R_q, i.e., up to $n^G_{i,q}$ times where $n^G_{i,q}$ is the number of τ_i's *gcs* in which it requests R_q. To calculate the total remote blocking we should calculate this type of blocking for any global resource request of τ_i. Therefore, the remote blocking is calculated as follows:

$$B_{i,3} = \sum_{\substack{R_q \in R^G_{P_k} \\ \wedge\, \tau_i \in \tau_{q,k} \\ \wedge\, P_k \in P_{A_i}}} n^G_{i,q}(\mathrm{RWT}_{q,A_i} + \sum_{\substack{P_l \in P_{A_i} \\ \wedge\, P_l \neq P_k}} z_{q,l}). \quad (7)$$

We can rewrite Equation 7 as follows:

$$B_{i,3} = \sum_{\substack{R_q \in R^G_{P_k} \\ \wedge\, \tau_i \in \tau_{q,k} \\ \wedge\, P_k \in P_{A_i}}} \alpha_{i,q}(\mathrm{RWT}_{q,A_i} + \sum_{\substack{P_l \in P_{A_i} \\ \wedge\, P_l \neq P_k}} z_{q,l}), \quad (8)$$

where $\alpha_{i,q} = n^G_{i,q}$.

Based on all three blocking terms introduced to a task τ_i in the system, the total blocking time of τ_i is as follows:

$$B_i = B_{i,1} + B_{i,2} + B_{i,3}. \quad (9)$$

According to Equation 8, it can be seen that the remote blocking of a task is a function of resource waiting time of its related application, i.e., the total blocking of a task is a function of resource waiting times of its corresponding application. Therefore we can rewrite Equation 9 as follows:

$$B_i = \gamma_i + \sum_{\substack{R_q \in R^G_{P_k} \\ \wedge\, \tau_i \in \tau_{q,k} \\ \wedge\, P_k \in P_{A_i}}} \alpha_{i,q}(\mathrm{RWT}_{q,A_i} + \delta_i), \quad (10)$$

where $\delta_i = \sum_{\substack{P_l \in P_{A_i} \\ \wedge\, P_l \neq P_k}} z_{q,l}$ and $\gamma_i = B_{i,1} + B_{i,2}$.

We note that δ_i and γ_i are only dependent on application internal parameters.

6.2 Requirements extraction for the application interface

In this section we extract the requirements Q_{A_i} for the interface of an application A_i from the schedulability analysis.

Each requirement in Q_{A_i} specifies a criteria of maximum resource wait times of one or more global resources from applications other than A_i in the system. We denote $mtbt_i$ as the maximum blocking time that τ_i can tolerate without missing its deadline. By definition, τ_i (scheduling according to FPS) is schedulable if:

$$0 < \exists t \leq D_i \;\; \mathtt{rbf}_{\mathsf{FP}}(i,t) \leq t, \quad (11)$$

where $\mathtt{rbf}_{\mathsf{FP}}(i,t)$ identifies the maximum cumulative execution requests that can be generated from the time that τ_i is released up to time t, which is implied as the *request bound function* of task τ_i and is computed as follows:

$$\mathtt{rbf}_{\mathsf{FP}}(i,t) = C_i + B_i + \sum_{\rho_i < \rho_j} (\lceil t/T_j \rceil C_j). \quad (12)$$

The maximum total blocking time that can be imposed on τ_i without missing its deadline is called $mtbt_i$ and it can be calculated using Equation 12 and substituting B_i by $mtbt_i$ as shown below:

$$mtbt_i = \max_{0<t\leq T_i} (t - (C_i + \sum_{\rho_i < \rho_j} (\lceil t/T_j \rceil C_j))). \quad (13)$$

Note that, it is not required to test all possible values for t in Equation 13, and only a bounded number of values for t that change $\mathtt{rbf}_{\mathsf{FP}}(i,t)$ should be considered (see [24] for more details). The total blocking time of task τ_i is a function of maximum resource wait times of the global resources accessed by tasks in its related application A_i. According to Equations 10 and 13 we can extract the requirement related to task τ_i as follows:

$$\gamma_i + \sum_{\substack{R_q \in R^G_{P_k} \\ \wedge\, \tau_i \in \tau_{q,k} \\ \wedge\, P_k \in P_{A_i}}} \alpha_{i,q}(\mathrm{RWT}_{q,A_i} + \delta_i) \leq mtbt_i. \quad (14)$$

therefore the related requirement to task τ_i will be as follows:

$$r_i \equiv \sum_{\substack{R_q \in R^G_{P_k} \\ \wedge\, \tau_i \in \tau_{q,k} \\ \wedge\, P_k \in P_{A_i}}} \alpha_{i,q}\mathrm{RWT}_{q,A_i} \leq mtbt_i - \gamma_i - \theta_i. \quad (15)$$

where $\theta_i = \sum_{R_q \in R^G_{P_k} \wedge\, \tau_i \in \tau_{q,k} \wedge\, P_k \in P_{A_i}} \alpha_{i,q}\delta_i$.

During the integration phase of applications, the schedulability of each application is tested using its requirements. An application A_i is schedulable if all the requirements in Q_{A_i} are satisfied. Note that in the requirements in Q_{A_i} the maximum resource wait time of A_i, RWT_{q,A_i}, for any global resource that is accessed by tasks within A_i, is calculated based on Equation 1.

7. APPLICATION PARTITIONING

One important challenge for the application developer is to partition the application on a given number of cores/processors. For resource constrained systems, the number of cores assigned for the system can be limited and it is required to use as few cores as possible. We propose semi-partitioned scheduling approach as an alternative for application-level partitioning. The motivation behind suggesting the semi-partitioned approach as a design choice for partitioning is shown by a simple example as follows:

Example. Assume a processor P_1 in a system where tasks are identified by the (C_i, T_i) model. Assume two tasks τ_1 and τ_2 with execution time and period of $(C+\varepsilon, 2C)$ where ε is an infinitesimal value (less than 1) and $(C, 2C)$ respectively. The utilization of τ_1 is $50\% + \varepsilon$ while the utilization of τ_2 is 50%. If we allocate τ_1 to P_1, then we can not allocate τ_2 to P_1 as well, since P_1's utilization exceeds 1. Therefore we have to add another processor, P_2 to which we can allocate τ_2. Now if we want to have another task τ_3 with similar execution and period of task τ_1, we can fit it on neither of the P_1 and P_2 processors due to the same reason. Hence, if we use the partitioned approach, then we should add a new processor to allocate τ_3. However with the semi-partitioned approach we can split the task in two parts and fit τ_3 on the combination of P_1 and P_2.

By the example above, it can be seen that the semi-partitioned approach may utilize the resources in a better way compared to the partitioned approach. However, without knowing the impact of resource sharing from other applications on an application under development, we can not decide how to allocate/partition the application such that all tasks will meet their deadlines. Also, depending on the system parameters, it might be enough to use the partitioned scheduling instead of the semi-partitioned approach so that the application is schedulable. On the other hand, selecting semi-partitioning as the design choice for the allocation algorithm and the choice which tasks should be split and how much should be split makes the search space very huge. Therefore, to increase the possibilities of finding a solution we suggest to use multiple interfaces for each application due to the possible use of both partitioned and semi-partitioned approaches for applications and we investigate the impact that the respective partitioning technique will have in the application interface.

As illustrated above, the semi-partitioned approach may utilize resources in a better way, but to provide more flexibility for the system designer we provide interfaces for both partitioned and semi-partitioned designs. As explained previously, there can be many different options to use the semi-partitioned approach as an alternative for application partitioning. We call each possible option, a configuration which will be discussed in Section 8.2. Each configuration can generate a different interface as will be seen later. However, the system developer will not know which configuration is better in terms of global system-level schedulability before the integration phase since the remote blocking from other applications is not available beforehand. Therefore, we propose to provide multiple interfaces for each application. The developer of the multiprocessor system can then select among the suggested interfaces, which have been extracted according to different partitioning designs, for the one that makes the whole platform schedulable.

8. MULTIPLE INTERFACE CONFIGURATION

For the sake of presentation simplicity and clarity, we use a simple case of an application allocated on two processors to illustrate the different interface configurations. Next we investigate the needed updates for an application interface according to different partitioning designs.

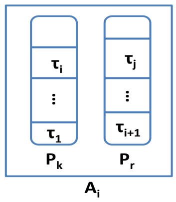

Figure 2: Application partitioning based on the partitioned approach

8.1 Partitioned Interface

As it can be seen in Figure 2, the task set τ_{A_i} related to an application A_i has been partitioned on processors P_k and P_r, such that a set of $(\tau_1, ..., \tau_i)$ tasks is allocated to P_k and a set of $(\tau_{i+1}, ..., \tau_j)$ tasks is allocated to P_r. For the sake of presentation simplicity and clarity, we assume that τ_i is the highest priority task on P_k and that τ_j is the highest priority task on P_r and both processors share the same set of global resources: $(R_1, ..., R_u)$. Based on these assumptions, the elements of A_i's interface, assuming the partitioned scheduling approach, as $I_{A_i}(Q_{A_i}, Z_{A_i})$ are specified as follows:

$$Q_{A_i} = \{Q_1, ..., Q_{\tau_i}, Q_{\tau_{i+1}}, ..., Q_{\tau_j}\} \tag{16}$$

$$Z_{A_i} = \{Z_{q_1,k}, ..., Z_{q_u,k}, Z_{q_1,r}, ..., Z_{q_u,r}\} \tag{17}$$

8.2 Semi-Partitioned Interface

Based on the semi-partitioned approach, two scenarios might be considered for the above mentioned example of two processors, as it can be seen in Figure 3 and Figure 4, where the highest priority task of each processor in the partitioned approach in Section 8.1 are the tasks that are split between two processors in each scenario. The reason of selecting the highest priority task to be split is that it has a great effect on the schedulability of all lower priority tasks within the systems. As it can be seen in Figure 3, τ_i is the task that is split on P_k and P_r such that τ_i^1 fills the capacity of P_k up to the allowed limit and τ_i^2, which has the remainder execution of τ_i, is located on P_r, [22, 23].

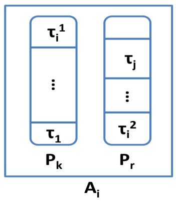

Figure 3: First scenario for application partitioning based on the semi-partitioned approach

Another scenario is where τ_j is the task that is split on P_k and P_r such that τ_j^1 fills the capacity of P_r up to the allowed limit, while τ_j^2 is located on P_k, as it can be seen in Figure 4.

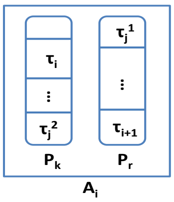

Figure 4: Second scenario for application partitioning based on the semi-partitioned approach

We assume that $\dot{I}_{A_i}(\dot{Q}_{A_i}, \dot{Z}_{A_i})$ and $\ddot{I}_{A_i}(\ddot{Q}_{A_i}, \ddot{Z}_{A_i})$ are the interfaces of application A_i under scenario 1 and scenario 2 respectively:

$$\dot{Q}_{A_i} = \{\dot{Q}_1, ..., \dot{Q}_{\tau_i^1}, \dot{Q}_{\tau_i^2}, ..., \dot{Q}_{\tau_j}\} \tag{18}$$

$$\dot{Z}_{A_i} = \{\dot{Z}_{q_1,k}, ..., \dot{Z}_{q_u,k}, \dot{Z}_{q_1,r}, ..., \dot{Z}_{q_u,r}\} \tag{19}$$

and

$$\ddot{Q}_{A_i} = \{\ddot{Q}_1, ..., \ddot{Q}_{\tau_i}, ..., \ddot{Q}_{\tau_j^1}, \ddot{Q}_{\tau_j^2}\} \tag{20}$$

$$\ddot{Z}_{A_i} = \{\ddot{Z}_{q_1,k}, ..., \ddot{Z}_{q_u,k}, \ddot{Z}_{q_1,r}, ..., \ddot{Z}_{q_u,r}\} \tag{21}$$

The presented analysis to evaluate the interface of each application for the case of the partitioned scheduling should be adapted when using semi-partitioned scheduling. Note that for the tasks that are allocated statically and are not split, they will not have further effect on the analysis that we presented in the previous section. However, the split tasks will have an impact on the analysis. The reason is that when a task in a subsystem is split, all its resource requests become global. It means that when using semi-partitioning the set of global resources requested by an application may change. Based on this, the resource hold time of tasks can vary, Equation 4, which results in changing the application locking time for each of its requests presented in the application interface. Furthermore, splitting a task, its execution time is also split in the corresponding cores which can affect the resulting interface of their applications. In addition, as mentioned above a critical section may occur at any time during the split task execution, therefore it may happen that a subtask which is within its global critical section has to migrate to its next processor, while locking the resource. However, we want to prevent this case to keep the same analysis as the analysis of the partitioned scheduling presented in this paper to find applications interfaces. This is done by letting the split tasks to overrun until they release the global resource then they are allowed to migrate to the next core [22, 23]. The overrun part should be considered in Equation 13.

To investigate the effect of partitioning on the interface parameters, we investigate how the split tasks affect the interface parameters. According to the first scenario, the subtask τ_i^2 is added to processor P_r while τ_i decreases its value of execution time in τ_i^1 on P_k. In P_k, all requirements will be affected by decreasing the execution time of task τ_i to τ_i^1, since in Equation 13 τ_i^1 as a higher priority task will affect *mbtb* of any task with priority lower than that of τ_i, and in this case include all tasks on P_k except τ_i itself. However, the requirement of task τ_i also changes, since the execution time of τ_i according to Equation 13 differs (in this case decreases). Subsequently, a change in Equation 13 results in a change of the requirement, due to Equation 14.

Similar changes happens also on P_r, since one task τ_i^2 is added to the processor which will affect *mbtb* of any task which is of lower priority than that of τ_i^2, as well as adding one extra requirement for τ_i^2.

Similar results can also be concluded under the second scenario with the difference that τ_j is decreasing to τ_j^1 on P_r, while τ_j^2 is added as an extra task to P_k. Therefore, we can conclude that:

$$Q_1 \neq \dot{Q}_1, ..., Q\tau_i \neq \dot{Q}_{\tau_i^1} \neq \dot{Q}_{\tau_i^2}, ..., Q_{\tau_j} \neq \dot{Q}_{\tau_j} \tag{22}$$

$$Q_1 \neq \ddot{Q}_1, ..., Q\tau_i \neq \ddot{Q}_{\tau_i}, ..., Q_{\tau_j} \neq \ddot{Q}_{\tau_j^1} \neq \ddot{Q}_{\tau_j^2} \tag{23}$$

The key challenge in interface extraction in the semi-partitioned approach is the requirement extraction, since some tasks are split among processors such as τ_i and τ_j in the first and second scenario. In order to extract the requirement of any task in the system we first have to specify the value of *mbtb* according to Equation 13 and then, by applying it in Equation 14, we extract the requirement. For the split task model, the deadline in Equation 13 for each subtask is the summation of the maximum response times of the previous subtasks [22, 23]. However, the worst-case response time of a task, requires the knowledge of the total blocking time duration, that is not provided during the application development. Therefore, for extracting the requirement for a subtask, we can assume explicitly the value of the deadline of each subtask of the split task. One possible way to do this can be by dividing the deadline of the task to equally for all subtasks, i.e., D_i/m, where D_i is the deadline of the original split task τ_i and m is the number of cores that τ_i is split among. This design choice helps the developer of the application to be able to abstract an application allocated to multiple processors under a semi-partitioned approach. Other options can be through using some weight based on the execution time of each subtask and/or the load in each core.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we develop a solution to integrate independently-developed real-time applications which may require more than one core/processor to be schedulable on a shared multi-core platform. We abstract each application resource demand including sharing mutually exclusive resources such that all internal tasks are schedulable via an interface. Therefore, by utilizing the information from the interfaces of other applications in the system, the schedulability of an application can be determined without performing schedulability analysis in task level. We have also suggested two design choices of partitioned and semi-partitioned techniques for application partitioning among processors. These suggested partitioning techniques provide a design method based on multiple interfaces for each application for better exploring the possibilities to find feasible solutions for application integration.

In the future, we plan to elaborate the addressed concerns related to the semi-partitioned approach to explore better solutions for application abstraction. Furthermore, we want to extend the solution for the case where applications can share processors/cores.

10. REFERENCES

[1] 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'11)*, Jul. 2011, pp. 251–261.

[2] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, dec. 2003, pp. 2 – 13.

[3] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "SIRAP: a synchronization protocol for hierarchical resource sharing in real-time open systems," in *Proceedings of the 7th ACM & IEEE conference on Embedded software (EMSOFT'07)*, October 2007, pp. 279–288.

[4] A. Easwaran, M. Anand, and I. Lee, "Compositional analysis framework using EDP resource models," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, dec. 2007, pp. 129 –138.

[5] J. Anderson, V. Bud, and U. Devi, "An EDF-based scheduling algorithm for multiprocessor soft real-time systems," in *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, Jul. 2005, pp. 199–208.

[6] X. Feng and A. Mok, "A model of hierarchical real-time virtual resources," in *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, 2002, pp. 26 – 35.

[7] G. Lipari and S. Baruah, "Efficient scheduling of real-time multi-task applications in dynamic systems," in *Real-Time Technology and Applications Symposium, 2000. RTAS 2000. Proceedings. Sixth IEEE*, 2000, pp. 166 –175.

[8] 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'07)*, Jul. 2007, pp. 247–258.

[9] I. Shin, A. Easwaran, and I. Lee, "Hierarchical scheduling framework for virtual clustering of multiprocessors," in *2008. ECRTS '08. 20th Euromicro Conference on Real-Time Systems*, july 2008, pp. 181 –190.

[10] R. Rajkumar, L. Sha, and J. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *Real-Time Systems Symposium (RTSS'88)*, Dec. 1988, pp. 259–269.

[11] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *10th International Conference on Distributed Computing Systems*, may/jun 1990, pp. 116–123.

[12] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *Computers, IEEE Transactions on*, vol. 39, no. 9, pp. 1175 –1185, sep 1990.

[13] 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 *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)*, May 2003, pp. 189–198.

[14] T. Baker, "Stack-based scheduling of real-time processes," *Journal of Real-Time Systems*, vol. 3, no. 1, pp. 67–99, 1991.

[15] 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'07)*, Aug. 2007, pp. 47–56.

[16] 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'08)*, Aug. 2008, pp. 185–194.

[17] ——, "Optimality results for multiprocessor real-time locking," in *31st IEEE Real-Time Systems Symposium (RTSS'10)*, Dec. 2010, pp. 49–60.

[18] S. Kato and N. Yamasaki, "Portioned static-priority scheduling on multiprocessors," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, april 2008, pp. 1 –12.

[19] ——, "Semi-partitioned fixed-priority scheduling on multiprocessors," in *15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'09)*, Apr. 2009, pp. 23–32.

[20] K. Lakshmanan, R. Rajkumar, and J. Lehoczky, "Partitioned fixed-priority preemptive scheduling for multi-core processors," in *21st Euromicro Conf. on Real-Time Sys. (ECRTS'09)*, Jul. 2009, pp. 239–248.

[21] N. Guan, M. Stigge, W. Yi, and G. Yu, "Fixed-priority multiprocessor scheduling with Liu and Layland's utilization bound," in *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10)*, Apr. 2010, pp. 165–174.

[22] S. Afshar, F. Nemati, and T. Nolte, "Towards resource sharing under multiprocessor semi-partitioned scheduling," in *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12), Work-in-Progress (WiP) session*, Jun. 2012.

[23] ——, "Resource sharing under multiprocessor semi-partitioned scheduling," in *18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'12)*, Aug. 2012.

[24] E. Bini and G. Buttazzo, "The space of rate monotonic schedulability," in *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, 2002, pp. 169 – 178.