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 the 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 each application 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 between processors and execute on any available processor. Semi-partitioned
scheduling is the 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. The reason of focusing on partitioned approaches as the de facto design choice is that they have been widely used in the industry and supported by commercial real-time operating systems [6]. In this paper we investigate the partitioned and semi-partitioned approaches to partition applications 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:

- Targeting 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 applications 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 requirements 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, 7, 8, 2]. Hierarchical scheduling techniques have been introduced and developed as a solution for these subsystems. Hierarchical scheduling has also been studied for multiprocessors [9, 10]. 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) [11] for shared memory multiprocessors. In DPCP a job accesses its local resources and executes 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. [11, 12], which is an extension of the Priority Ceiling Protocol (PCP) [13] 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. [14], which extends the Stack Resource Policy (SRP) [15] 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. [16] for both partitioned and global scheduling, which later was extended to partitioned FMLP by Brandenburg and Anderson [17]. 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 [18] 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 [19, 20, 21, 22], 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 [22]. 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 [23, 24], 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 within applications may differ from each other 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 \( A_i \) is denoted by \( \tau_{A_i} \) and consists of \( n \) sporadic hard real-time tasks \( \tau_i(\tau_i, C_i, D_i, p_i) \), where \( \tau_i \) identifies the minimum inter-arrival time between two successive jobs of task \( \tau_i \) with worst-case execution time \( C_i \) and priority \( p_i \). \( D_i \) 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 \( A_i \) share a set of resources \( R_{A_i} \) which are protected using semaphores. The set of shared resources \( R_{A_i} \) con-
exists of two subsets of different types of resources; local and global resources. Local resources are shared by 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 \( R^L_k \) and \( R^G_k \) respectively, i.e., \( R_k = R^L_k \cup R^G_k \). We denote \( C_{i,q} \) as the worst-case execution time of the longest critical section in which a task \( \tau_i \) 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 \( \tau_i \) as a subset of \( l \) subtasks \( \tau_i^1, \ldots, \tau_i^l \), and each subtask is represented by \( (C_{ij}, T_i, D_i, R_i, O_i^j) \) where \( (k = 1, \ldots, l) \). \( O_i^j \) represent the constant offset of the \( k \)th subtask of the split task \( \tau_i \) 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 \( \tau_i \), \( T_i, D_i \) and \( R_i \) are the same as \( \tau_i \) [24].

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 [24].

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 task-level information of 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 \( P_{Ai} \). We first specify some definitions and terms needed in this context.

4.1 Resource Hold Time
\( RHT_{q,k} \) is the resource hold time and it defines the maximum time duration that the global resource \( R_q \) can be held by \( \tau_i \) executing on \( P_k \) [1]. By definition, \( RHT_{q,k} \) accounts for the longest critical section in which \( \tau_i \) 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 \( R_q \) is presented by \( Z_{q,k} \) and denotes the maximum duration of time that any task on processors other than \( P_k \) that may request \( R_q \) 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 \( R_q \) denoted by \( Z_{A_i} \) 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 \( P_l \) requesting a 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 \( P_l \) for global resource \( R_q \) and is presented by \( RWT_{q,l} \). 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 \( I_{Ai}(Q_{Ai}, Z_{Ai}) \), where \( Q_{Ai} \) 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 the platform becomes schedulable.

On the other hand, each application in the system may introduce 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. \( Z_{Ai} \) in the interface of \( A_i \) represents these delays which is a set \( Z_{q,Ai} \) for each global resource \( R_q \) requested by \( A_i \).

5. GENERAL DESCRIPTION
For each global resource in the system a unique FIFO queue is dedicated in which the applications associated to tasks inside the application requesting the resource are enqueued whenever the request is not satisfied. Note that, since the applications are independently developed, the relative priority among tasks in different applications is not defined which makes the use of FIFO queue for global resources 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, \( \tau_i \) is requesting a global resource \( R_q \) and since the resource is not available the request is inserted in the related global FIFO queue as shown in the picture. \( \tau_i \) will be suspended since the request is not on the head of the global FIFO queue. \( A_i \) is the application allocated on processors \( P_1 \) and \( P_2 \) and \( A_j \) is the application on processors \( P_3 \) and each processor consists of a set of tasks 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 τᵢ granted access to a global resource is immediately boosted to the value equal to \( pᵢ + pₘᵦₖ(Pᵢ) \), where \( pₘᵦₖ(Pᵢ) = \max \{ pⱼ | τⱼ ∈ Pᵢ \} \). It means that the task can preempt all higher priority tasks which do not use any global resource and all lower priority tasks no matter they are using a global shared resource or not. This cause the blocking times of tasks to become a function of global critical sections (gcs) only.

Rule 3: When a task τᵢ running on processor Pᵢ within application Aᵢ requests a global resource Rᵢ, τᵢ access the resource if the resource is available (i.e., Rᵢ’s FIFO queue is empty) otherwise, a placeholder for the request of τᵢ associated to Aᵢ is added to the resource FIFO queue of Rᵢ and τᵢ is suspended.

Rule 4: When a global resource Rᵢ becomes available to the application Aᵢ, 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 τᵢ on processor Pᵢ inside application Aᵢ releases a global resource Rᵢ, then the placeholder of Aᵢ will be removed from the resource queue and the resource becomes available to the processor whose application is at the head of Rᵢ’s queue. Also, the priority of the task will return to its original priority.

6. APPLICATION ANALYSIS

In this section we extract the needed elements in interface of an application which will enable the system schedulability analysis. We assume that applications use partitioned scheduling and in Section 7 we will investigate how the analysis can be adjusted under semi-partitioned scheduling. First we elaborate the resource hold time and resource wait time of the tasks in the application level.

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 Aᵢ has to wait for Rᵢ to be available for Aᵢ occurs when all other applications in the system has requested Rᵢ just before Aᵢ and as a consequence are already waiting in the FIFO queue of Rᵢ. Therefore, the resource wait time for Aᵢ based on the interference from other applications ZₐᵢAᵢ is calculated as follows:

\[
RWT_{q,Aᵢ} = \sum_{all Aⱼ ∉ Aᵢ} ZₐⱼAᵢ. \tag{1}
\]

According to ZₐⱼAᵢ’s definition along with the resource handling queue structure which is FIFO based, the maximum blocking time that Aᵢ can introduce to any task τⱼ in an application other than Aᵢ occurs when all tasks in all processors of Aⱼ that share Rⱼ request Rⱼ just before τⱼ and their requests enqueued in the FIFO queue before τⱼ’s request. Note that, the longest time that Rⱼ can be locked by processor Pⱼ is \( Zⱼ₁Kⱼ \). Therefore the application locking time of Aᵢ on a global resource Rⱼ is calculated as follows:

\[
ZₐᵢAᵢ = \sum_{Pⱼ ∈ Pᵢ} Zⱼ₁Kⱼ. \tag{2}
\]

As it can be seen, if Aᵢ is allocated on one processor, then ZₐᵢAᵢ becomes similar to \( Zⱼ₁Kⱼ \).

\( Zⱼ₁Kⱼ \) is the maximum blocking time imposed by all tasks τⱼKⱼ that share Rⱼ and are located on Pⱼ, on other tasks in other processors, e.g., τᵢ. The maximum \( Zⱼ₁Kⱼ \) happens when these tasks τⱼKⱼ request Rⱼ before τᵢ. The maximum time that Rⱼ can be locked by each element in τⱼKⱼ is by definition RHTⱼKⱼ. Thus the processor locking time of Pⱼ on Rⱼ is calculated as follows:

\[
Zⱼ₁Kⱼ = \sum_{τᵢ ∈ τⱼKⱼ} RHTᵢKⱼ. \tag{3}
\]

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

\[
RHTᵢKⱼ = Cₛᵢⱼ + HᵢKⱼ. \tag{4}
\]

where \( HᵢKⱼ \) denotes the interference from higher priority tasks, which is calculated as follows [1]:

\[
HᵢKⱼ = \sum_{pᵢ < pⱼ} \sum_{Rⱼ ∈ RⱼG} Cₛᵢⱼ. \tag{5}
\]

6.1 Blocking Terms

In this section we describe the possible scenarios where a task τᵢ 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 τᵢ is blocked on a global resource, it gives the chance to a lower priority task τⱼ to lock a local resource, which in turn may block τᵢ when it resumes after releasing the global resource. We represent the number of gcs’s of τᵢ by \( n₉G \). The above mentioned scenario can happen up to \( n₉G \) times. In addition, according to local synchronization protocols such as PCP and SRP, task τᵢ can be blocked on a local resource by at most one critical section of a lower priority task which has arrived before τᵢ. On the other hand, τⱼ can release a maximum of \( [Tᵢ/Tⱼ] \) jobs before τᵢ’s current job is finished. Furthermore, each job of τⱼ can block τᵢ’s current job at most \( n₉L(Tᵢ) \) times, where \( n₉L(Tᵢ) \) denotes the number of critical sections in which τᵢ requests local resources with ceiling higher than that of priority τᵢ. Therefore, the blocking time on local resources, which is denoted by \( B_{l,1} \), upper bounds as follows:

\[
B_{l,1} = \min \{ n₉L(Tᵢ) + 1 \}, \sum_{pᵢ < pⱼ} \sum_{Rⱼ ∈ RⱼG} [Tᵢ/Tⱼ]n₉L(Tᵢ) \} \max \left\{ \frac{Cₛᵢⱼ}{pᵢ < pⱼ} \right\} \left\{ Rⱼ ∈ RⱼG \right\} \left\{ Rⱼ ≤ \text{ceil}(Rᵢ) \right\}. \tag{5}
\]

where \text{ceil}(Rᵢ) = \max \{ pᵢ | τᵢ ∈ τⱼKⱼ \}.

6.1.2 Local blocking due to global resources

Each time τᵢ suspends on a global resource, a lower priority task τⱼ may access a global resource which subsequently can preempt τᵢ after it resumes and finishes its gcs in its non-gcs sections. This situation may also happen when τⱼ arrives sooner than τᵢ. Therefore, this blocking can happen as many times as τᵢ are requesting global resources up to \( n₉G \) times in addition to the case where τⱼ may arrive sooner than τᵢ, which causes a maximum of \( n₉G + 1 \) times.

Similar to the previous case, τⱼ can release at most \( [Tᵢ/Tⱼ] \) jobs before τᵢ’s current job finishes. On the other hand, each job of τⱼ can preempt τᵢ’s current job a maximum of \( n₉G \) times.
Thus, this kind of blocking introduced by $\tau_j$ to $\tau_i$ denoted by $B_{i,2}$ can happen at most $\min\{n^G_i + 1, \lceil T_i/T_j \rceil n^G_j\}$ times which is upper bounded as follows:

$$B_{i,2} = \sum_{\rho_j \in \rho_i} \sum_{\tau_j \leq \tau_i} \left( \min\{n^G_i + 1, \lceil T_i/T_j \rceil n^G_j\} \max_{R_q \in R_q^\rho_i} \{C_{s_j,q}\} \right). \quad (6)$$

### 6.1.3 Remote blocking

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

$$B_{i,3} = \sum_{R_q \in R_q^i} n^G_i (RWT_{q,t}A_i + \sum_{\rho_j \in \rho_i} \sum_{\tau_j \leq \tau_i} z_{q,t}^i). \quad (7)$$

We can rewrite Equation 7 as follows:

$$B_{i,3} = \sum_{R_q \in R_q^i} n^G_i (RWT_{q,t}A_i + \sum_{\rho_j \in \rho_i} \sum_{\tau_j \leq \tau_i} z_{q,t}^i), \quad (8)$$

where $\alpha_{q,t} = n^G_i$.

Based on all three blocking terms introduced to a task $\tau_i$ in the system, the total blocking time of $\tau_i$ 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:

$$B_i = \gamma_i + \sum_{R_q \in R_q^i} n^G_i (RWT_{q,t}A_i + \delta_i), \quad (10)$$

where $\delta_i = \sum_{\rho_j \in \rho_i} \sum_{\tau_j \leq \tau_i} z_{q,t}^i$ and $\gamma_i = B_{i,1} + B_{i,2}$.

We note that $\delta_i$ and $\gamma_i$ 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 $\text{mtbt}_i$ as the maximum blocking time that $\tau_i$ can tolerate without missing its deadline. By definition, $\tau_i$ (scheduling according to FPS) is schedulable if:

$$0 < \exists r \leq D_i \text{ rbf}_{FP}(i,t) \leq t, \quad (11)$$

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

$$\text{rbf}_{FP}(i,t) = C_i + B_i + \sum_{p_j < P_i} \left(\lceil T_i/T_j \rceil C_j\right). \quad (12)$$

The maximum total blocking time that can be imposed on $\tau_i$ without missing its deadline is called $\text{mtbt}_i$ and it can be calculated using Equation 12 and substituting $B_i$ by $\text{mtbt}_i$ as shown below:

$$\text{mtbt}_i = \max_{0 \leq t \leq D_i} \left(t - C_i + \sum_{p_j < P_i} \left(\lceil T_i/T_j \rceil C_j\right)\right). \quad (13)$$

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 $\text{rbf}_{FP}(i,t)$ should be considered (see [25] for more details). The total blocking time of task $\tau_i$ 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 $\tau_i$ as follows:

$$\gamma_i + \sum_{R_q \in R_q^i} \sum_{\tau_j \leq \tau_i} \alpha_{q,t} (RWT_{q,t}A_i + \delta_i) \leq \text{mtbt}_i. \quad (14)$$

therefore the related requirement to task $\tau_i$ will be as follows:

$$r_i = \max_{0 \leq t \leq D_i} \left(t - C_i + \sum_{p_j < P_i} \left(\lceil T_i/T_j \rceil C_j\right)\right). \quad (15)$$

where $\theta_i = \sum_{R_q \in R_q^i} \sum_{\tau_j \leq \tau_i} \alpha_{q,t} \delta_i$. 

"
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 $A_i$, $RW_{T_{A_i}}$, 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 developers 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 $\tau_1$ and $\tau_2$ with execution time and period of $(C + \varepsilon, 2C)$ where $\varepsilon$ is an infinitesimal value (less than 1) and $(C, 2C)$ respectively. The utilization of $\tau_1$ is $50\% + \varepsilon$ while the utilization of $\tau_2$ is $50\%$. If we allocate $\tau_1$ to $P_1$, then we cannot allocate $\tau_2$ 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 $\tau_2$. Now if we want to have another task $\tau_3$ with similar execution and period of task $\tau_1$, 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 $\tau_3$. However with the semi-partitioned approach we can split the task in two parts and fit $\tau_1$ 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. In other words, the tasks within an application which are schedulable standalone may become unschedulable when integrating with other applications. Also, depending on the system parameters, it might be enough to use the partitioning 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 multi-processor 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.

![](image)

Figure 2: Application partitioning based on the partitioned approach

8.1 Partitioned Interface

As it can be seen in Figure 2, the task set $\tau_1$, related to an application $A_1$, has been partitioned on processors $P_1$ and $P_2$, such that a set of $(\tau_1, \ldots, \tau_3)$ tasks is allocated to $P_1$ and a set of $(\tau_4, \ldots, \tau_5)$ tasks is allocated to $P_2$. For the sake of presentation simplicity and clarity, we assume that $\tau_1$ is the highest priority task on $P_1$ and that $\tau_4$ is the highest priority task on $P_2$ and both processors share the same set of global resources: $(R_1, \ldots, R_w)$. 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:

$$Q_{A_i} = \{Q_1, \ldots, Q_r, Q_{r+1}, \ldots, Q_j\} \quad (16)$$

$$Z_{A_i} = \{Z_{q_0}, \ldots, Z_{q_w}, Z_{r_0}, \ldots, Z_{r_w}\} \quad (17)$$

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 impact on the schedulability of all lower priority tasks within the systems. As it can be seen in Figure 3, $\tau_i$ is the task that is split on $P_k$ and $P_j$, such that $\tau_i$ fills the capacity of $P_k$ up to the allowed limit and $\tau_j$ which has the remainder execution of $\tau_i$ is located on $P_j$, [23, 24].

Another scenario is where $\tau_j$ is the task that is split on $P_k$ and $P_j$ such that $\tau_j$ fills the capacity of $P_j$ up to the allowed limit, while $\tau_i$ is located on $P_k$, as it can be seen in Figure 4.
and

Equation 13. They release the global resource then they are allowed to migrate to interfaces. This is done by letting the split tasks to overrun until prevent this case to keep the same analysis as the analysis of the next processor, while locking the resource. However, we want to task which is within its global critical section has to migrate to its which can affect the resulting interface of their applications. In ad-

quests presented in the application interface. Furthermore, splitting a task, its execution time is also split in the corresponding cores to \(Pr_1\) since one task \(\tau_j^2\) is added to the processor which will affect \(mbtb\) of any task which is of lower priority than that of \(\tau_j^1\), as well as adding one extra requirement for \(\tau_j^2\).

Similar results can also be concluded under the second scenario with the difference that \(\tau_j\) is decreasing to \(\tau_j^1\) on \(P_1\), while \(\tau_j^1\) is added as an extra task to \(P_2\). Therefore, we can conclude that:

\[Q_1 \neq Q^{rc1}_1, \ldots, Q_{r_i} \neq Q^{rc1}_1, \ldots, Q_{r_i} \neq \tau_j^{rc1}, \ldots, Q_{r_i} \neq \tau_j^{rc1} \]  \hspace{1cm} (22)

\[Q_1 \neq Q^{rc2}_1, \ldots, Q_{r_i} \neq Q^{rc2}_1, \ldots, Q_{r_i} \neq \tau_j^{rc2}, \ldots, Q_{r_i} \neq \tau_j^{rc2} \]  \hspace{1cm} (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 \(\tau_j\) and \(\tau_j^1\) 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 [23, 24]. 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 \(\tau_i\) and \(m\) is the number of cores that \(\tau_i\) 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 schedu-

able 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 task-level schedulability analysis. 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.

Acknowledgments
The work presented in this paper has been supported by The Swedish Foundation for Strategic Research, The Swedish Research Council, The University via Mäalardalen Real-time Research Center (M-RTC) at Mälardalen University.

10. REFERENCES