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

- Targeting independently-developed applications that are al-
located 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 par-
titioning configurations, providing flexibility and better re-
source 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 subsys-
tem abstraction by assuming partitioned and semi-partitioned ap-
proaches 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 en-
vironment 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 sup-
port sharing of mutually exclusive resources. In the context of re-
source 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 Proto-
col (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 shar-
ing 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 Flexi-
ble 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 appli-
cations. 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/ sub-
systems are developed independently and abstracted in their inter-
faces, 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 Mono-
tonic Scheduling (RMS) for an arbitrary task set [21]. In these
works, tasks are assumed to be independent, i.e., no resource shar-
ing is allowed between tasks.

Inspired by our previous work on supporting resource sharing un-
der semi-partitioned scheduling [22, 23], and based on the subsys-
tem 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 plat-
form in the remainder of the paper, is composed of identical pro-
cessors with shared memory. An application consists of a task set
and a particular scheduling algorithm and tasks may request mutu-
ally 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 appli-
cations may differ between applications, e.g., one application may
use a Fixed Priority Scheduling (FPS) policy, while another appli-
cation may apply a dynamic priority scheduling policy (e.g., Ear-
liest Deadline First EDF). However, due to space limitations and
presentation clarity, in this paper we assume only the usage of FPS.
A task of an application A_k is denoted by τ_{ik} and consists of n
sporadic hard real-time tasks τ_i(T_i,C_i,D_i,p_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 p_i, D_i represents
the task’s deadline where D_i ≤ T_i. We also assume that each task in
an application has a unique priority.

The tasks on application P_i share a set of resources R_{P_i} which are
protected using semaphores. The set of shared resources R_{P_i} con-
ists 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_i as R_{P_i}^L and R_{P_i}^G respectively, i.e., R_{P_i} = R_{P_i}^L ∪ R_{P_i}^G.
We denote C_{i,q} as the worst-case execution time of the longest critical
section in which a task τ_{iq} 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 \( \{ C_k^i, T_k, D_k, p_k, O_k^i \} \) where \( (k = 1, \ldots, l) \). \( O_k^i \) represent the constant offset of the \( k \)th subtask of split task \( \tau_i \) which is identified by the former subtask’s maximum response time. The offset of the first subtask is zero, \( O_1^i = 0 \). For the subtasks of a split task \( \tau_i \), \( T_k, D_k \) and \( p_k \) are the same as \( \tau_i \) [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

\( RHT_{q,k,i} \) is the resource hold time and it defines the maximum time duration that the global resource \( R_q \) can be held by \( \tau_i \) executing on \( P_k \) [1]. By definition, \( RHT_{q,k,i} \) 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 \( R_q \) 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 \( RW_{q,k} \). 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 \( RW_{q,A_i} \).

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](image)

Figure 1 shows a simple example of a system that consists of two applications running on 3 cores. At a certain time, \( \tau_1 \) 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. \( \tau_1 \) will be suspended since the request is not on the head of the global FIFO queue. \( P_2 \), \( P_3 \) and \( P_5 \) 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 \( \tau_i \) granting access to a global resource is immediately boosted to the value equal to \( p_\tau + p_{\text{max}}(P_k) \), where \( p_{\text{max}}(P_k) = \max\{ p_\tau | \tau \in P_k \} \). 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 \( \tau_i \) located on processor \( P_k \) related to an application \( A_i \) requests a global resource, \( \tau_i \) access the resource if the resource is available (i.e., the global resource FIFO queue is empty) otherwise, a request for \( \tau_i \) associated to \( A_i \) is added to the resource FIFO queue and \( \tau_i \) 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 \( \tau_j \) 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 \( Z_{q,A_i} \) is calculated as follows:

\[
RWT_{q,A_i} = \sum_{\forall A_j, j \neq A_i} Z_{q,A_j}. \quad (1)
\]

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 \( \tau_i \) 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 \( \tau_i \) and their requests enqueued in the FIFO queue before \( \tau_i \)'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:

\[
Z_{q,A_i} = \sum_{P_l \in P_k} Z_{q,k}. \quad (2)
\]

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 \( \tau_{q,k} \) that share \( R_q \) and are located on \( P_k \) on other tasks in other processors, e.g., \( \tau_i \). The maximum time that \( R_q \) can be locked by each element in \( \tau_{q,k} \) is by definition \( RHT_{q,k,i} \). Thus the processor locking time of \( P_k \) on \( R_q \) is calculated as follows:

\[
Z_{q,k} = \sum_{\tau_i \in \tau_{q,k}} RHT_{q,k,i}. \quad (3)
\]

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

\[
RHT_{q,k,i} = CS_{i,q} + H_{i,q,k}. \quad (4)
\]

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

\[
H_{i,q,k} = \sum_{\rho \in \rho_i \wedge R_k \in R_{q,k} \wedge \rho \neq q} CS_{j,k}.
\]

6.1 Blocking Terms

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

\[
B_{i,j} = \min \{ n_i^j + 1, \sum_{\rho < \rho_i \wedge R_k \in R_{q,k}} \lceil \frac{T_j}{T_f} \rceil \} \max \{ CS_{j,k} \}, \quad (5)
\]

where \( ceil(R_l) = \max \{ \rho | \tau_i \in \tau_{j,k} \} \).

6.1.2 Local blocking due to global resources

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

Similar to the previous case, \( \tau_j \) can release at most \( \lceil \frac{T_i}{T_f} \rceil \) jobs before \( \tau_i \)'s current job finishes. On the other hand, each job of \( \tau_j \) can preempt \( \tau_j \)'s current job a maximum of \( n_i^j \) times.

Thus, this kind of blocking introduced by \( \tau_j \) to \( \tau_i \), denoted by \( B_{i,j} \), can happen at most \( \min \{ n_i^j + 1, \lceil \frac{T_i}{T_f} \rceil \} \) times which is upper bounded as follows:

\[
B_{i,j} = \sum_{\rho < \rho_i \wedge \tau_i \in \tau_{j,k}} \left( \min \{ n_i^j + 1, \lceil \frac{T_i}{T_f} \rceil \} \max \{ CS_{j,k} \} \right). \quad (6)
\]
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 $\tau_j$ on processor $P_k$ belonging to application $A_j$ 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 $\tau_i$, 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 $n^G_{i,q}$ times where $n^G_{i,q}$ 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:

$$B_{i,3} = \sum_{R_q \in R^G_i} n^G_{i,q} \cdot (RWT_{q,A_i} + \sum_{P_j \in P_{A_i} \land R_j \neq P_k} \alpha_{q,j}) \cdot \tau_i.$$  

(7)

We can rewrite Equation 7 as follows:

$$B_{i,3} = \sum_{R_q \in R^G_i} \alpha_{q,i} \cdot (RWT_{q,A_i} + \sum_{P_j \in P_{A_i} \land R_j \neq P_k} \alpha_{q,j}) \cdot \tau_i.$$  

(8)

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

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:

$$B_i = B_{i,1} + B_{i,2} + B_{i,3}.$$  

(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:

$$B_i = \gamma_i + \sum_{R_q \in R^G_i} \alpha_{q,i} \cdot (\text{RWT}_{q,A_i} + \delta_i).$$  

(10)

where $\delta_i = \sum_{P_j \in P_{A_i}} \sum_{R_j \in P_k} \tau_{ij}$ 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 $Q_A$ for the interface of an application $A_i$ from the schedulability analysis.

Each requirement in $Q_A$ 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 $\tau_i$ can tolerate without missing its deadline. By definition, $\tau_i$ (scheduling according to FPS) is schedulable if:

$$0 < \exists t \leq D_i \ rbf_{FP}(i,t) \leq t.$$  

(11)

where $\ rbf_{FP}(i,t)$ 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:

$$\ rbf_{FP}(i,t) = C_i + B_i + \sum_{P_j \neq P_i} ([t/T_j]C_j).$$  

(12)

The maximum total blocking time that can be imposed on $\tau_i$ 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:

$$mtbt_i = \max_{0 \leq t \leq T_i} (t - C_i + \sum_{P_j \neq P_i} ([t/T_j]C_j)).$$  

(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 $\ rbf_{FP}(i,t)$ should be considered (see [24] 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:

$$\gamma_i + \sum_{R_q \in R^G_i} \alpha_{q,i} \cdot (\text{RWT}_{q,A_i} + \delta_i) \leq mtbt_i.$$  

(14)

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

$$r_i \equiv \sum_{R_q \in R^G_i} \alpha_{q,i} \cdot \text{RWT}_{q,A_i} \leq mtbt_i - \gamma_i - \theta_i.$$  

(15)

where $\theta_i = \sum_{R_q \in R^G_i} \alpha_{q,i} \delta_i$.

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 $Q_A$ are satisfied. Note that in the requirements in $Q_A$, the maximum resource wait time of $A_i$, $\text{RWT}_{q,A_i}$, 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 $(C, T_i)$ model. Assume two tasks $\tau_1$ and $\tau_2$ with execution time and period of $(C+\epsilon, 2C)$ where $\epsilon$ is an infinitesimal value (less than 1) and $(C, 2C)$ respectively. The utilization of $\tau_1$ is 50% + $\epsilon$ while the utilization of $\tau_2$ is 50%. If we allocate $\tau_1$ to $P_1$, then we can not 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 $\tau_3$ 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](image)

**8.1 Partitioned Interface**

As it can be seen in Figure 2, the task set $\tau_A$, related to an application $A_1$ has been partitioned on processors $P_k$ and $P_r$, such that a set of $\tau_{1j}, ..., \tau_{ij}$ tasks is allocated to $P_k$ and a set of $\tau_{i+1j}, ..., \tau_{ij}$ tasks is allocated to $P_r$. For the sake of presentation simplicity and clarity, we assume that $\tau_1$ is the highest priority task on $P_k$ and that $\tau_j$ is the highest priority task on $P_r$ and both processors share the same set of global resources: $(R_1, ..., R_{ka})$. Based on these assumptions, the elements of $A_j$’s interface, assuming the partitioned scheduling approach, as $I_A(Q_{A_1}, Z_{A_1})$ are specified as follows:

$$Q_{A_1} = \{Q_1, ..., Q_{c_1}, Q_{c_1+1}, ..., Q_{c_j}\}$$

$$Z_{A_1} = \{Z_{q_1+k}, ..., Z_{q_1+k}, Z_{q_1+r}, ..., Z_{q_1+r}\}$$

**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, $\tau_i$ is the task that is split on $P_k$ and $P_r$ such that $\tau_i$ fills the capacity of $P_k$ up to the allowed limit and $\tau_i^2$, which has the remainder execution of $\tau_i$, is located on $P_r$.

![Figure 3: First scenario for application partitioning based on the semi-partitioned approach](image)
Another scenario is where $\tau_j$ is the task that is split on $P_k$ and $P_j$ such that $\tau_j^1$ fills the capacity of $P_j$ up to the allowed limit, while $\tau_j^2$ 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](image)

We assume that $I_{A_j}(Q_{A_j}, Z_{A_j})$ and $I_{A_j}(Q_{A_j}, Z_{A_j})$ are the interfaces of application $A_j$ under scenario 1 and scenario 2 respectively:

$$Q_{A_j} = \{Q_1, \ldots, Q_{\tau_j^1}, Q_{\tau_j^2}, \ldots, Q_{\tau_j}\} \quad (18)$$

$$Z_{A_j} = \{Z_{\tau_j}, k, \ldots, Z_{\tau_j}, k, Z_{\tau_j}, r, \ldots, Z_{\tau_j}, r\} \quad (19)$$

and

$$Q_{A_j} = \{Q_1, \ldots, Q_{\tau_j}, Q_{\tau_j}, \ldots, Q_{\tau_j}, \tau_j^2\} \quad (20)$$

$$Z_{A_j} = \{Z_{\tau_j}, k, \ldots, Z_{\tau_j}, k, Z_{\tau_j}, r, \ldots, Z_{\tau_j}, r\} \quad (21)$$

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 $\tau_j^1$ is added to processor $P_j$ while $\tau_j$ decreases its value of execution time in $\tau_j^2$ on $P_k$. In $P_k$, all requirements will be affected by decreasing the execution time of task $\tau_j$ to $\tau_j^2$, since in Equation 13 $\tau_j$ as a higher priority task will affect mbtb of any task with priority lower than that of $\tau_j$, and in this case include all tasks on $P_k$ except $\tau_j$ itself. However, the requirement of task $\tau_j$ also changes, since the execution time of $\tau_j$ 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 $P_j$, 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 $\tau_j^2$, 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 $P_j$, while $\tau_j^1$ is added as an extra task to $P_j$. Therefore, we can conclude that:

$$Q_1 \neq Q_1, \ldots, Q_{\tau_j} \neq Q_{\tau_j}, Q_{\tau_j} \neq Q_{\tau_j}, \ldots, Q_{\tau_j} \neq Q_{\tau_j} \quad (22)$$

$$Q_1 \neq Q_1, \ldots, Q_{\tau_j} \neq Q_{\tau_j}, \ldots, Q_{\tau_j} \neq Q_{\tau_j} \quad (23)$$

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 $\tau_j$ 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., $D_l/m$, where $D_l$ is the deadline of the original split task $\tau_j$ and $m$ is the number of cores that $\tau_j$ 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