A Hierarchical Scheduling Model for Component-Based Real-Time Systems

José L. Lorente
Universidad de Cantabria
lorentejl@unican.es

Giuseppe Lipari
Scuola Superiore Sant'Anna
lipari@sssup.it

Enrico Bini
Scuola Superiore Sant'Anna
e.bini@sssup.it

Abstract

In this paper, we propose a methodology for developing component-based real-time systems based on the concept of hierarchical scheduling. Recently, much work has been devoted to the schedulability analysis of hierarchical scheduling systems, in which real-time tasks are grouped into components*, and it is possible to specify a different scheduling policy for each component. Until now, only independent components have been considered.*

In this paper, we extend this model to tasks that interact through remote procedure calls. We introduce the concept of abstract computing platform *on which each component is executed. Then, we transform the system specification into a set of real-time* transactions *and present a schedulability analysis algorithm. Our analysis is a generalization of the holistic analysis to the case of abstract computing platforms. We demonstrate the use of our methodology on a simple example.*

1 Introduction

The research on design methodologies for real-time embedded systems is now moving towards component-based approaches. This trend can be explained with the need for a structured approach to rapid development of increasingly complex systems, and for short time-to-market requirements. Among many advantages of component-based approaches, we cite the possibility to easily reuse existing pieces of software that provide a well-defined functionality; a better de-composition of a complex system into separate, smaller and more manageable sub-systems; the possibility to upgrade the system by adding or replacing components. However, a component-based methodology for real-time embedded systems covering all aspects of the design flow is still to come. One of the main problems is that a component specification must include functional as well as non-functional parameters, like periods and deadlines, and allow a *compositional* schedulability analysis of the entire system starting from the components properties.

Many design methodologies for real-time systems have been proposed in the literature. One of the most popular is HRT-HOOD [3], which has been recently extended to UML [9]. A system is specified as a set of *active*, *protected* or *pas-sive* objects, and it can be analyzed with classical fixed priority schedulability analysis. Similar design methodologies have been proposed in the context of UML. In particular, the Object Management Group (OMG) has proposed a *Profile for Schedulability, Performance and Time* (SPT) [5]. However, schedulability analysis can only be performed after all the system components have been specified.

Recently, several approaches to the problem have been proposed that are based on *hierarchical scheduling* [12, 4, 1, 7]. In such approaches, the system consists of many independent *applications* (also referred to as *components* in [12]). An *application* is a set of periodic or sporadic tasks and a local dedicated scheduler. Each application is assigned a fraction of the computational resource. The system implements a two-level scheduling strategy: at the *global level*, a system-wide scheduler selects which application is to be scheduled next. The application then selects which task to execute by running the local scheduler. This approach guarantees *temporal* isolation: the behavior of an application does not depends on the characteristics of the other applications in the system.

The hierarchical scheduling approach has many advantages. First, the global scheduler needs no information on the internal characteristics of the applications. Thanks to this separation of concerns, it is possible to change the internal implementation of the application (even the local scheduler) without influencing the rest of the system. Second, thanks to the temporal isolation property, it is possible to analyze each application independently from the rest of the system. For these reasons, the hierarchical scheduling approach can be seen as a promising starting point for a component-based methodology [12, 4, 8].

Until now, applications have been considered independent from each other. This is a very strong limitation for applying such techniques to real component-based systems. Moreover, given the recent trends in embedded real-time systems, it is necessary to take into account multiprocessor and distributed systems.

Contributions of this paper In this paper, we provide a first step to the final objective of a component-based real-time methodology. We follow the hierarchical scheduling approach by defining a component as a set of real-time tasks plus a real-time scheduler, but unlike previous work,

we allow components to interact through Remote Procedure Calls (RPC). We start by defining the component interface as a set of *provided* and *required methods*, and the component implementation as a set of real-time threads (Section 2). We also specify how to integrate a set of components into the final system architecture. Then, we describe our scheduling architecture by defining the concept of *abstract computing platform* (in Section 2.3) on which each component executes. Finally, given a set of components, we provide a schedulability analysis to see if the response time of every thread is within the thread's deadline. Our analysis is a generalization of the holistic analysis [10, 14] to the case of abstract computing platforms.

2 System model

In this section we give informal definitions of the building blocks of our methodology. The concepts will be explained by means of a simple example that will be carried out through the entire paper. In our model, a system consists of a set of *components* that are connected through a *system architecture*. Each component executes upon an *abstract computing platform*.

2.1 Components

A component consists of a *provided interface*, a *required interface*, and an *implementation*.

The *provided interface* is a set of methods offered to other components of the system. Each method is characterized by: (1) the method **signature**, which is the name of the method and the list of parameters, and (2) a **worst-case activation pattern**, which describes the maximum number of invocations the method is able to handle in an interval of time. In this paper, we will restrict this parameter to a single value, the *minimum interarrival time* (MIT) between two consecutive calls of the method.

We use the dot notation, common in object oriented languages, to identify the different parameters of a component. For example, the method `read()` of the provided interface of component `A` is denoted by `A.provided.read` and its MIT by `A.provided.read.T`.

The semantic of invocation of a method can be synchronous (the caller waits for the method to be completed) or asynchronous (the caller continues to execute without waiting for the completion of the operation). In this paper we will only consider synchronous method invocation.

The *required interface* is a set of methods that the component requires for carrying out its services. Each method is characterized by its signature and the MIT, as in the provided interface. Method `write` of component `A`'s required interface will be denoted by `A.required.write`.

The *implementation* of a component is the specification of how the component carries out its work. In our model, a component is implemented by a set of concurrent threads

and by a scheduler. The scheduler is *local* to a component, and is used to schedule the component threads. Different components can have different schedulers. In this paper, we will only consider a fixed priority local scheduler. However, our methodology can be easily extended to other local schedulers like EDF. Using local schedulers allows independent analysis of each component. The thread priorities are *local* to a component: the schedulability analysis of the component can be carried out earlier in the design cycle with obvious advantages from a design point of view.

Threads can be activated in two different ways: (1) **time-triggered**, when it is activated periodically, and (2) **event-triggered**, when it is activated by a call to a provided method of the component.

Time-triggered threads are assigned a period and a relative deadline, denoted by `Thread.T` and `Thread.D`, respectively. The parameters of the event-triggered threads derive from the corresponding parameters coming from the provided method to which they are attached. See Section 2.4 for more details.

Each thread is then implemented by a sequence of *tasks* and *method calls*. Tasks are pieces of code that are implemented directly by the component, while method calls are invocations of methods of the required interface. As anticipated before, in this paper we only consider synchronous method invocations: when invoking a method of the required interface, which is realized by a provided method of a different component, the thread is suspended until the method is completed.

2.2 Example

We now present an example of a very simple subsystem. The goal of this system is to realize a sensor fusion between the readings of two sensors of the same kind located in different places in the environment. As an example, consider a stereoscopic system which tries to compute the distance from an object in the environment. We decompose the application into 3 components: two components will get the images and extract the relevant features, while a third component will analyze the results of the two feature extractions and compute the distance of a specified object.

The `SensorReading` components are two realization of the same component class described in Figure 1. The interfaces and the implementation are specified in pseudo object oriented languages.

The component provides a method `read()` that can be invoked by other components to obtain the value of the most recent sensor data. The component is self-contained, as it does not require any method from other components. It implements its functionality with 2 threads scheduled by a fixed priority scheduler. `Thread1` is periodic with period equal to 15 msec, and has the highest priority. Its goal is to periodically get the data from the sensor board. `Thread2`

```
SensorReading {
provided:
  read();        // MIT = 50 msec
implementation:
  Scheduler: fixed priority;
  Thread1 : periodic(period=15 msec),
           priority=2;
  Thread2 : realizes read(), priority = 1;
}
```

Figure 1. The `SensorReading` class.

```
SensorIntegration {
provided:
  read();
required:
  readSensor1();
  readSensor2();
implementation:
  Thread1 : realizes read(), priority=1;
  Thread2 : periodic(period=50 msec),
           priority=2
  {
    init;
    readSensor1();
    readSensor2();
    compute();
  }
}
```

Figure 2. The `SensorIntegration` class.

realizes the interface to the other components: it is invoked every time another component calls the `read()` method, and has the lowest priority. We do not report the code of the two threads.

The second component class implements the sensor integration component, and is reported in Figure 2. The provided interface consists on a single method `read()` which allows other components to read the sensor data. Since this component has to merge the data read from the sensors, its required interface consists of two methods `readSensor1()` and `readSensor2()`. The implementation consists of two threads: `Thread1` acts as an interface towards the other components and realizes method `read()`; `Thread2` periodically reads from the two sensors and performs the elaboration. The pseudo-code of `Thread2` is reported in the ﬁgure. The period of `Thread2` is 50 msec, and its relative deadline is the same as the period.

2.2.1 Integration

The ﬁnal subsystem under analysis consists of two components of class `SensorReading` and one component of class `SensorIntegration`, called , `Sensor1`, `Sensor2` and `Integrator`, respectively. The required interface of `Integrator` will be *connected* to the provided interfaces of `Sensor1` and `Sensor2`.

We now need to specify what happens when a method

of a component is invoked. Suppose a component A invokes a method of another component B. If the two components reside in the same computational node the method invocation is just a function call with no delay. However, if the two components are located on different computational nodes the invocation of the method results in one or more messages being sent over a network. Thus, we assume that abstraction of RPC is realized by some middleware or operating system mechanism. From a schedulability analysis point of view, we assume that the network is similar to a computational node and messages are scheduling according to the network scheduling policy [2].

2.3 Computing platforms

In our model, each component is executed on a dedicated *abstract computing platform*. Each abstract computing platform runs on top of a *physical computing platform*. The mechanisms that implements the abstract platforms upon the physical platform is the *global scheduler*.

One possible strategy for global scheduling is to use an aperiodic server algorithm like Polling Server, CBS or similar [7, 1, 12]. Moreover, it is also possible to use different strategies, like static partitioning of the resource [4] or a p-fair scheduler [13], depending on the available physical platform. The important thing is that the underlying scheduling mechanism supports the concept of reservation: each component is *reserved* a fraction of the physical computing platform. From the components point of view, the shared physical platform can then be seen a set of independent abstract computing platforms. This approach is valid for both the processor and the network.

The concept of abstract computing platform does not rely on any speciﬁc global scheduling policy. In fact, it is simply characterized by the amount of cycles it can provide to the components in any arbitrary interval of time. The modeling through the provided cycles has been already adopted both in networking [6, 2] and in real-time systems [7, 12, 1]. We denote the abstract computing platform by the letter Π.

The actual computational resource that is provided depends clearly on the speciﬁc conditions which occurs online (i.e. depending on the requirements of the component). However we can generalize this concept by deﬁning the minimum and the maximum number of cycles which can be provided by a platform Π. For this purpose we deﬁne the *supply functions*.

Definition 1 *Let* Π *be a computing platform, we deﬁne* minimum supply function $Z_{\Pi}^{\min}(t)$ *of the platform, the function deﬁned as follows*

$$Z_{\Pi}^{\min}(t) = \min_{t_0}\{cycles\ provided\ in\ [t_0, t_0 + t]\ under$$

$$any\ conditions.\} \quad (1)$$

Definition 2 *Let Π be a computing platform, we de ne maximum supply function $Z_\Pi^{\max}(t)$ of the platform, the function de ned as follows*

$$Z_\Pi^{\max}(t) = \max_{t_0}\{cycles\ provided\ in\ [t_0, t_0 + t]\ under$$

$$any\ conditions.\} \quad (2)$$

The amount of cycles provided by the platform Π in the interval $[t_0, t_0 + t]$, for any t_0, is always between $Z_\Pi^{\min}(t)$ and $Z_\Pi^{\max}(t)$.

Let us provide an example in order to clarify the meaning of the two functions. Suppose that a computing platform can provide Q cycles every period P. As reported in Figure 3, the minimum amount of cycles are experienced when the beginning of the interval $[t_0, t_0 + t]$ coincides with the end of a previous time quanta Q and the next time quanta is delayed as much as possible. For the same reason the maximum computational resource is provided when we obtain the time quanta Q as we require it, and the next period P begins right when we have finished the quanta, so that a new amount of time Q is available. The two scenarios
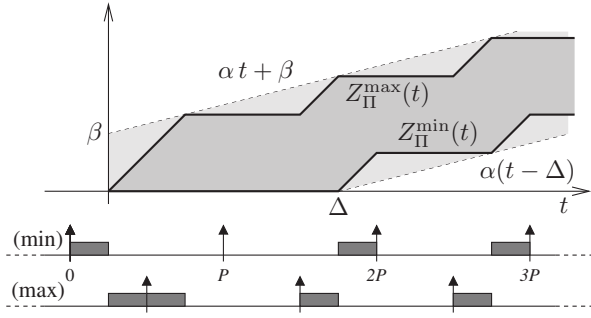

Figure 3. Examples of $Z_\Pi(t)$.

are depicted in the Figure 3, where the best scenario is labelled with "(max)" and the worst scenario is labelled with "(min)".

The actual number of provided cycles will depend on the on-line conditions. However the two functions $Z_\Pi^{\max}(t)$ and $Z_\Pi^{\min}(t)$ allows to bound the number of cycles provided in any interval $[t_0, t_0 + t]$, meaning that the actual supply function is always in the darker gray region of Figure 3.

The figure shows the max/min supply function of a periodic server. In general the mechanism behind the computing platform Π may be more complex. If, for example, Π is implemented by a pfair task the min/max supply functions will be quite different than before. However, it is still possible to calculate the min/max supply functions.

The first very important characteristic of the computing platform is related to the notion of *rate*. Intuitively the rate is the speed at which the computing platform can provide the computational power. We define it formally as follows.

Definition 3 *Let Π be a computing platform and $Z_\Pi^{\min}(t)$,*

$Z_\Pi^{\max}(t)$ *the minimum and the maximum supply functions respectively. We de ne* maximum rate *of the platform Π the following quantity:*

$$\alpha^{\max} = \lim_{t \to +\infty} \frac{Z_\Pi^{\max}(t)}{t} \quad (3)$$

and the minimum rate

$$\alpha^{\min} = \lim_{t \to +\infty} \frac{Z_\Pi^{\min}(t)}{t}. \quad (4)$$

If $\alpha^{\max} = \alpha^{\min}$ then we simply call this value rate *of the platform and we indicate it with α.*

Notice that in the periodic task model (Figure 3) we have $\alpha^{\min} = \alpha^{\max} = \frac{Q}{P}$. All the state-of-art mechanisms implementing a computing platform have the minimum rate equal to maximum rate. For this reason, in this paper we always assume this condition and we will only refer to the rate α.

The rate of a platform can be interpreted as the average slope of the supply functions. We can find linear upper and lower bounds of the supply function.

Definition 4 *Let Π be a computing platform and α its rate. We de ne the* delay Δ *of Π the following*

$$\Delta = \max\{d \geq 0 : \exists t \geq 0 \quad Z_\Pi^{\min}(t) \leq \alpha(t - d)\}. \quad (5)$$

Definition 5 *Let Π be a computing platform and α its rate. We de ne the* burstiness β *of Π the following*

$$\beta = \max\{b \geq 0 : \exists t \geq 0 \quad Z_\Pi^{\max}(t) \geq b + \alpha t\}. \quad (6)$$

The two additional parameters Δ and β, characterizing a computing platform, are called respectively delay and burstiness for their analogy with the network calculus [6].

In this paper we suppose that a set of M computing platforms is available, and we model the jth platform Π_j by the triple $(\alpha_j, \Delta_j, \beta_j)$. It is very important to remark that this model is a generalization of the classical analysis. In fact, by setting $\alpha = 1$, $\Delta = 0$ and $\beta = 0$ we obtain a processor used at its full capacity.

Finally we also highlight that the cost of using a general model is payed in terms of the pessimism introduced estimating the supply function by linear functions.

2.4 Deriving transactions from components

We model each periodic thread as a *real-time transaction*. A transaction Γ_i is a sequence of tasks $\tau_{i,j}$. So we have $\Gamma_i = (\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,n_i})$, with a precedence constraint, i.e. task $\tau_{i,j}$ cannot start before task $\tau_{i,j-1}$ has completed.

Each transaction is characterized by a period T_i and a relative deadline D_i, meaning that the last task τ_{i,n_i} must complete no later than D_i after the first task of the transaction has been activated.

Each task $\tau_{i,j}$ is characterized by a worst-case execution time $C_{i,j}$, a best-case execution time $C_{i,j}^{\text{best}}$, an offset $\overline{\phi}_{i,j}$ (which is the first instant from the activation of the transaction at which the task could be activated), a jitter $J_{i,j}$
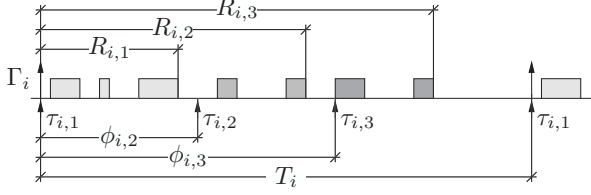
Figure 4. Transaction and task parameters

which is the maximum delay computed from the offset after which the task can be activated, and a mapping variable $s_{i,j} \in \{1, 2, \ldots, M\}$ which means that the task is allocated on the platform $\Pi_{s_{i,j}}$. These are the classical parameters used in the holistic analysis [14] and their relationship is depicted in Figure 4. The worst-case response time of a task $\tau_{i,j}$ is denoted by $R_{i,j}$ and is measured from the activation time of the corresponding transaction Γ_i. The priority of task $\tau_{i,j}$ is denoted by $p_{i,j}$. A greater $p_{i,j}$ corresponds to a higher priority. We allow both the offset $\overline{\phi}_{i,j}$ and jitter $J_{i,j}$ to be larger than period of its transactions T_i. In order to simplify the notation, we consider a reduced task offset, $\phi_{i,j} = \overline{\phi}_{i,j} \mod T_i$, which is always within in $[0, T_i)$.

Given these definitions, we transform the set of all threads in the system into a set of transactions. Remember that, according to the definitions in Section 2.1, a thread consists of a sequence of one or more tasks (i.e. pieces of code or internal procedures) and zero or more invocations of external methods.

We start from a periodic thread belonging to some component. All tasks belonging to the thread will be part of one transaction; moreover, if a method invocation is part of the thread, all the tasks belonging to the thread corresponding to the method call will be part of the transaction. The algorithm is applied recursively until all periodic threads are transformed into transactions.

All the tasks are assigned priorities equal to the priorities of the threads they belong to (remember that in this paper we are assuming a fixed priority local scheduler inside each component). The transaction is assigned a period and a relative deadline equal to the period and relative deadline of the originating periodic thread.

The algorithm is better explained through an example. Consider again the simple system described in Section 2.2. `Integrator.Thread2` is a periodic thread that will originate the first transaction Γ_1. It consists of a task `init` (that we rename $\tau_{1,1}$), plus two method invocations `readSensor1()` and `readSensor2()`. These methods are connected to threads `Sensor1.Thread1` and `Sensor2.Thread1`. Therefore, they will be part of the transaction as well. Since these two threads do not invoke any other methods and consist of only one task, they originate two tasks $\tau_{1,2}$ and $\tau_{1,3}$. Finally, `Integrator.Thread1` includes another task `compute` that we rename as $\tau_{1,4}$. All other periodic threads in the sys-

tem do not invoke any other method and consist of only of one task. Therefore, they generate transactions Γ_2, Γ_3 and Γ_4 with one task each.

In general messages can simply be modeled by considering additional "tasks" that have to be "executed" on a abstract computing platform that models the network. However, for the sake of clarity, in this example we ignored the messages exchanged between the components for realizing the remote procedure call.
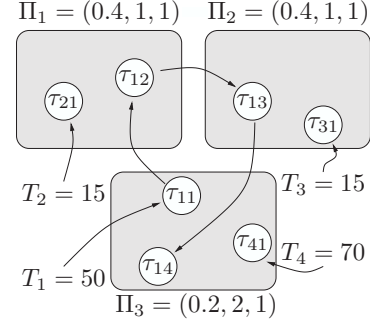

Figure 5. Example of application.

3 Analysis on abstract computing platforms

At this point we are ready to analyze the schedulability of a system consisting of real-time transactions, where different tasks of a transaction can execute on different abstract computing platforms.

In this paper, we assume that all components implement local fixed priority schedulers. To ensure the feasibility of the system, we need to compute the worst-case response time of the transaction and check whether this value lower than or equal to the relative deadline.

3.1 Analysis for Task with Static Offsets

We analyze a set of tasks executing onto the same computing platform Π. As seen in Section 2.4, the tasks are grouped in transactions. Now we assume that all the offsets $\phi_{i,j}$ and jitters $J_{i,j}$ of the task $\tau_{i,j}$ are known and static.

The proposed analysis extends several works on transaction scheduling [15, 10] when the computing platform Π is modelled by the triple (α, Δ, β) as seen in Section 2.3. This is equivalent to say that the computation times of all the tasks are scaled by a factor $\frac{1}{\alpha}$ and they suffer an additional blocking equal to Δ.

3.1.1 Exact Response Time Analysis

Now we show how to compute the worst-case response time of a generic task $\tau_{a,b}$. From now on we indicate the task under analysis by $\tau_{a,b}$. As proved by Tindell [15], the worst-case for a task happens inside a *busy period* for the task under analysis, where we adopt the same definition of busy period as in [15].

Without loss of generality, we set the beginning of the

busy period for task $\tau_{a,b}$ at time 0. Let us focus on the activation pattern of task $\tau_{i,j}$. We call Φ_i the length of the interval between the activation of the transaction Γ_i that occurred immediately before or at the beginning of the busy period, and the beginning of the busy period. Obviously, $0 \le \Phi_i < T_i$. For each task $\tau_{i,j}$ we define the phase $\varphi_{i,j}$ as the instant of its first activation after $t = 0$. Formally,

$$\varphi_{i,j}(\Phi_i) = T_i - (\Phi_i - \phi_{i,j}) \mod T_i \quad (7)$$

Thanks to previous results on transaction analysis (Theorem 1 in [10]), we know that the worst-case pattern of activation of the higher priority jobs occurs when all the jobs are released as close as possible to the start of the busy period.

Now, assume that the busy period of $\tau_{a,b}$ has length t. According to Theorem 1 in [10], the worst-case contribution of a higher priority task $\tau_{i,j}$ to the busy period is:

$$W_{i,j}(\Phi_i, t) = \left(\left\lfloor \frac{J_{i,j} + \varphi_{i,j}(\Phi_i)}{T_i} \right\rfloor + \left\lceil \frac{t - \varphi_{i,j}(\Phi_i)}{T_i} \right\rceil \right) \frac{C_{i,j}}{\alpha} \quad (8)$$

The total contribution of of the transaction Γ_i to the busy period is obtained by considering the contributions of all higher priority tasks, as follows

$$W_i(\Phi_i, t) = \sum_{\tau_{i,j} \in \mathsf{hp}_i(\tau_{a,b})} W_{i,j}(\Phi_i, t) \quad (9)$$

where $\mathsf{hp}_i(\tau_{a,b})$ is the set of tasks in Γ_i with priority greater than or equal to $p_{a,b}$.

The contribution of Γ_i to the busy period, as it is possible to see in Eq. (9), depends on the value of Φ_i. In order to evaluate the maximum response time of the task $\tau_{a,b}$, we must take into account many possible values of Φ_i. The next theorem [10] determines all these values.

Theorem 1 (Theorem 2 in [10]) *The worst-case contribution of transaction Γ_i to a task $\tau_{a,b}$ is obtained when the first activation of some task $\tau_{i,k}$ in $\mathsf{hp}_i(\tau_{a,b})$ that occurs within the busy period coincides with the critical instant, after having experienced the maximum possible delay, i.e., the maximum jitter, $J_{i,k}$.*

If we known the task $\tau_{i,k}$ starting the busy period of the transaction Γ_i, then Eq. (7) becomes:

$$\varphi_{i,j}(\Phi_i(\tau_{i,k})) = \varphi_{i,j}^k = T_i - (\phi_{ik} + J_{ik} - \phi_{i,j}) \mod T_i \quad (10)$$

where we introduced the compact notation of $\varphi_{i,j}^k$.

Using this value, we can now obtain the expression of the worst-case contribution Γ_i, when the busy period is initiated with $\tau_{i,k}$. We will call this function $W_i^k(\tau_{a,b}, t)$, and we obtain it by replacing Eq. (10) in Equation (8) and (9):

$$W_i^k(\tau_{a,b}, t) = \sum_{\tau_{i,j} \in \mathsf{hp}_i(\tau_{a,b})} \left(\left\lfloor \frac{J_{i,j} + \varphi_{i,j}^k}{T_i} \right\rfloor + \left\lceil \frac{t - \varphi_{i,j}^k}{T_i} \right\rceil \right) \frac{C_{i,j}}{\alpha} \quad (11)$$

In order to obtain the worst-case response time of task $\tau_{a,b}$ we need to evaluate the above function for all the transactions in the system. Unfortunately we don't know in ad-

vance which tasks $\tau_{i,k} \in \mathsf{hp}_i(\tau_{a,b})$ would lead to the maximum response time. For this reason we must consider all the possible scenarios. Each scenario is represented by a vector of indexes ν. $\nu(i)$ is associated to the transaction Γ_i and varies in $1, \ldots, n_i$. A scenario ν means that all the tasks $\tau_{i,\nu(i)}$ are released simultaneously at $t = 0$.

The number of possible scenarios $N(\tau_{a,b})$ is determined by the number of tasks in $\mathsf{hp}_i(\tau_{a,b})$ that exist in each transaction Γ_i. Moreover the task $\tau_{a,b}$ may be released at time $t = 0$ as well. Thus, the total number of scenarios is:

$$N(\tau_{a,b}) = (N_a(\tau_{a,b}) + 1) \prod_{\substack{i \ne a \\ \mathsf{hp}_i(\tau_{a,b}) \ne \varnothing}} N_i(\tau_{a,b}) \quad (12)$$

where $N_i(\tau_{a,b})$ is the number of tasks in $\mathsf{hp}_i(\tau_{a,b})$.

For convenience, we will number the jobs of the task under analysis using the letter p and we denote the job by $\tau_{a,b}(p)$. Also, the jobs are ordered consecutively meaning that $\tau_{a,b}(p)$ is activated in $((p-1)T_a, pT_a]$.

For each scenario ν we obtain the completion time of each job $\tau_{a,b}(p)$ in the busy period. This time, labelled $w_{a,b}^\nu(p)$, is obtained by considering the execution time of the jobs of $\tau_{a,b}$ together with the interference from all the higher priority tasks, plus the blocking time of the task and the platform delay Δ. It can be calculated by the following iterative formula:

$$w_{a,b}^{\nu,(n+1)}(p) = \Delta + B_{a,b} + (p - p_{0,a,b}^\nu + 1)\frac{C_{a,b}}{\alpha} + \sum_i W_{i\nu(i)}(\tau_{a,b}, w_{a,b}^{\nu,(n)}(p)) \quad (13)$$

where $p_{0,a,b}^\nu = 1 - \left\lfloor \frac{J_{a,b} + \varphi_{a,b}^{\nu(a)}}{T_a} \right\rfloor$.

Equation (13) can be solved by starting from a value of $w_{ab}^{\nu,(0)}(p) = 0$, and iterating until two consecutive iterations produce the same value. The maximum value of p that we need to check is

$$p_{L,a,b}^\nu = \left\lceil \frac{L_{a,b}^\nu - \varphi_{a,b}^{\nu(a)}}{T_a} \right\rceil \quad (14)$$

where $L_{a,b}^\nu$ is the length of the busy period, which may be obtained with the following iterative expression:

$$L_{a,b}^{\nu,(n+1)} = \left(\Delta + B_{a,b} + \left\lceil \frac{L_{a,b}^{\nu,(n)} - \varphi_{a,b}^{\nu(a)}}{T_a} \right\rceil - p_{0,a,b}^\nu + 1 \right) \frac{C_{a,b}}{\alpha} + \sum_i W_i^{\nu(i)}(\tau_{a,b}, L_{a,b}^{\nu,(n)})$$

The response time of the job $\tau_{a,b}(p)$ is obtained subtracting the instant at which the event activated the transaction from the obtained completion time. Therefore the activation of the pth job occurs at $\varphi_{a,b}^{\nu(a)} + (p-1)T_a - \overline{\phi}_{a,b}$. Consequently the response time for job $\tau_{a,b}(p)$ is:

$$R_{a,b}^\nu(p) = w_{a,b}^\nu(p) - (\varphi_{a,b}^{\nu(a)} + ((p-1)T_a - \overline{\phi}_{a,b})$$

To calculate the response time $R_{a,b}$ of the task $\tau_{a,b}$ we

must determine the maximum among all the possible scenarios. Then we have:

$$R_{a,b} = \max_{\nu} \max_{p \in \{p^{\nu}_{0,a,b}, \ldots, p^{\nu}_{L,a,b}\}} R^{\nu}_{a,b}(p)$$

Unfortunately, this approach is very complex because it considers all the possible scenarios ν. For this reason we apply the same approximation technique used in the literature [15, 10] to our case of abstract computing platform.

3.1.2 Reducing the number of scenarios

Tindell [15] observed that the interference of the transaction Γ_i on the task $\tau_{a,b}$ can be maximized with respect to all the tasks in $\mathsf{hp}_i(\tau_{a,b})$. This means that the interference can be certainly upper bounded by

$$W^*_i(\tau_{a,b}, t) = \max_{\tau_{i,k} \in \mathsf{hp}_i(\tau_{a,b})} W^k_i(\tau_{a,b}, t) \qquad (15)$$

In order to reduce the pessimism, we don't use the upper bound of Eq. (15) for the transaction Γ_a to which the task under analysis belongs. Instead we use the original Eq. (11) for Γ_a. Consequently, for this simpler analysis we have to consider only the scenarios created by the tasks in the set $\mathsf{hp}_a(\tau_{a,b}) \cup \tau_{ab}$. Assuming that the busy period is started by $\tau_{a,c} \in \mathsf{hp}_a(\tau_{ab}) \cup \tau_{a,b}$, then the response time is determined by the following iterative equation

$$w^{c(n+1)}_{a,b}(p) = \Delta + B_{a,b} + (p - p^c_{0,a,b} + 1)\frac{C_{a,b}}{\alpha} +$$
$$\sum_{i \neq a} W^*_i(\tau_{a,b}, w^{c(n)}_{a,b}(p)) + W^c_a(\tau_{a,b}, w^{c(n)}_{a,b}(p)) \quad (16)$$

The length of the busy period is then calculated iteratively by

$$L^{c(n+1)}_{a,b} = \Delta + B_{a,b} + \left(\left\lceil \frac{L^{(n)}_{a,b} - \varphi^c_{a,b}}{T_a} \right\rceil - p^c_{0,a,b} + 1 \right)\frac{C_{a,b}}{\alpha}$$
$$+ \sum_{i \neq a} W^*_i(\tau_{a,b}, L^{c(n)}_{a,b}) + W^c_a(\tau_{a,b}, L^{c(n)}_{a,b})$$

and from it we can compute $p^c_{L,a,b}$ using Eq. 14, which is the last job of $\tau_{a,b}$ within the busy period.

As done previously, we compute the response time of job $\tau_{a,b}(p)$, when $\tau_{a,c}$ is released at $t = 0$

$$R^c_{a,b}(p) = w^c_{a,b}(p) - (\varphi^c_{a,b} + ((p-1)T_a - \overline{\phi}_{a,b})$$

and then the upper bound of the response time is

$$R_{a,b} = \max_{\tau_{a,c} \in \mathsf{hp}_a(\tau_{a,b}) \cup \tau_{a,b}} \max_{p \in \{p^c_{0,a,b}, \ldots, p^c_{L,a,b}\}} (R^c_{a,b}(p))$$

The possibles scenarios examined in this case are represented by the set $\mathsf{hp}_a(\tau_{a,b}) \cup \tau_{a,b}$. The number of this scenarios now is $N_a(\tau_{a,b}) + 1$, which is significantly less than the number of scenarios of the exact analysis, reported in Eq. (12).

3.2 Analysis onto Abstract Computing Platforms

In this sections we extend the analysis presented previously to the abstract computing platform. Since the task $\tau_{a,b}$ under analysis runs on a specific computing platform $\Pi_{s_{a,b}}$,

only the tasks allocated on the same platform can interfere with it. For this reason we redefine the set of interfering tasks as follows:

$$\mathsf{hp}_i(\tau_{a,b}) = \{\tau_{i,j} \in \Gamma_i : p_{i,j} \geq p_{a,b} \wedge s_{i,j} = s_{a,b}\} \quad (17)$$

Also, we must replace the rate α and the delay Δ with $\alpha_{s_{a,b}}$ and $\Delta_{s_{a,b}}$, in order to consider the platform where the task is allocated.

In the transaction model the first task $\tau_{i,1}$ is released every period T_i. Every other task $\tau_{i,j}$ is released when the previous task $\tau_{i,j-1}$ is completed. Due to this observation, a relationship between the task response times, offsets and jitters can be established as follows

$$\overline{\phi}_{i,j} = R^{\text{best}}_{i,j-1}$$
$$J_{i,j} = R_{i,j-1} - R^{\text{best}}_{i,j-1} \qquad (18)$$

where $R^{\text{best}}_{i,j-1}$ and $R_{i,j-1}$ are respectively a lower bound to the best-case response time and a upper bound for the worst-case response time of the task $\tau_{i,j-1}$.

Using the offset and jitter terms defined in Eq. (18) and the set of interfering tasks as in Eq. (17), we can use the same iterative method that was presented in Section 3.1 for the case of static offsets.

The "static offset" iterative algorithm can be iterated at a higher level in order to find the response times of all the tasks, similarly as done in [10]. The initial values of jitters and offsets are $J_{i,j} = 0$ and $\overline{\phi}_{i,j} = R^{\text{best}}_{i,j}$ for each task. The convergence of the "dynamic offset" iterative algorithm is guaranteed by the monotonic dependency of the response times and the jitter terms.

Now we address the problem of calculating the best-case response time, which can provably speed up the whole iterative process. There are several techniques to calculate the best-case response time, $R^{\text{best}}_{i,j}$. The simplest one consists of considering only the shortest execution time of the preceding jobs of its transactions, which are $\{\tau_{i,1}, \ldots, \tau_{i,j-1}\}$. It follows that the best-case response time of each task is

$$R^{\text{best}}_{i,j} = \sum_{k=1}^{j-1} \max\left\{0, \frac{C^{\text{best}}_{i,k}}{\alpha_{s_{i,k}}} - \beta_{s_{i,k}}\right\}$$

where the term $\max\left\{0, \frac{C^{\text{best}}_{i,k}}{\alpha_{s_{i,k}}} - \beta_{s_{i,k}}\right\}$ is the best-case computation time of task $\tau_{i,k}$, when it runs onto an abstract computing platform. Notice that if the platform has a high value of burstiness, we experience a shorter best-case computation time, as expected.

Ola Redell at al. [11] proposed a method to calculate the exact best-case response time. Their method can be applied to our algorithm with no major changes.

Finally, by means of our analysis we can obtain the response time of every transaction. We assert that the system is schedulable if the last task of every transactions meets the deadline, meaning that $R_{i,n_i} \leq D_i$.

4 Example

In order to better understand the analysis presented, we illustrate it on the example of Section 2.2. Consider the system of Figure 5. Platform Π_1 contains tasks $\{\tau_{1,2}, \tau_{2,1}\}$, platform Π_2 contains tasks $\{\tau_{1,3}, \tau_{3,1}\}$ and platform Π_3 contains tasks $\{\tau_{1,1}, \tau_{1,4}\}$ (we remind that in this example we do not model messages, and therefore we assume 0 delay between tasks of the same transactions). The values of the platforms' and tasks' parameters are reported in the Tables 1 and 2.

Task	Platform	$C_{i,j}^{\text{best}}$	$C_{i,j}$	T_i	D_i	$p_{i,j}$	$\phi_{i,j}^{\min}$
$\tau_{1,1}$	Π_3	0.8	1	50	50	2	0
$\tau_{1,2}$	Π_1	0.8	1	50	50	1	3
$\tau_{1,3}$	Π_2	0.8	1	50	50	1	4
$\tau_{1,4}$	Π_3	0.8	1	50	50	3	5
$\tau_{2,1}$	Π_1	0.25	1	15	15	3	0
$\tau_{3,1}$	Π_2	0.25	1	15	15	3	0
$\tau_{4,1}$	Π_3	5	7	70	70	1	0

Table 1. Parameters of the example.

Platform	α_i	Δ_i	β_i
$\Pi_1 (Sensor\,1)$	0.4	1	1
$\Pi_2 (Sensor\,2)$	0.4	1	1
$\Pi_3 (Integrator\,3)$	0.2	2	1

Table 2. Parameters for the platforms.

Task	$J_{i,j}^{(0)}$	$R_{i,j}^{(0)}$	$J_{i,j}^{(1)}$	$R_{i,j}^{(1)}$	$J_{i,j}^{(2)}$	$R_{i,j}^{(2)}$	$J_{i,j}^{(3)}$	$R_{i,j}^{(3)}$	$J_{i,j}^{(4)}$	$R_{i,j}^{(4)}$
$\tau_{1,1}$	0	12	0	**12**						
$\tau_{1,2}$	0	9	9	18	9	**18**				
$\tau_{1,3}$	0	10	5	15	14	24	14	**24**		
$\tau_{1,4}$	0	12	5	17	10	22	19	39	19	**39**

Table 3. Results for transaction Γ_1

In Table 3 we report the results of the successive iterations of the analysis on the tasks of transaction Γ_1. The transaction is schedulable as the response time of $\tau_{1,4}$ is less than the transaction deadline (50 msec).

5 Conclusions and future works

In this paper, we presented a component-based design methodology for real-time embedded systems. Our methodology extends the concept of hierarchical scheduling [12, 4, 1, 7] to the case of components that interact through remote procedure calls. We further defined the concept of abstract computing platform, and proposed a schedulability analysis.

Many open problems need to be solved before the methodology can be considered for actual use. In this paper, we assumed that the parameters of the abstract computing platform are known. However, they could be computed depending on the actual requirement of a component. This requires an optimization method to assign the parameters (α, β, Δ) to each abstract platform. The search for the optimal platform parameters would allow a better utilization of the resources, and it will be the subject of our future work.

References

[1] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proceedings of the 4th ACM International Conference on Embedded software*, pages 95–103, Pisa, Italy, 2004. ACM Press.

[2] L. Almeida, P. Pedreiras, and J. A. G. Fonseca. The FTT-CAN protocol: Why and how. *IEEE Transaction on Industrial Electronics*, 49(6):1189–1201, Dec. 2002.

[3] A. Burns and A. Wellings. *HRT-HOOD: a structured design for hard real-time systems*. Elsevier Science, 1995.

[4] X. Feng and A. K. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 26–35, Austin, TX USA, Dec. 2002.

[5] O. M. Group. *UML Profile For Schedulability, Performance, And Time, Version 1.0*. http://www.omg.org, 2001.

[6] J.-Y. Le Boudec and P. Thiran. *Network Calculus*, volume 2050 of *Lecture Notes in Computer Science*. Springer, 2001.

[7] G. Lipari and E. Bini. A methodology for designing hierarchical scheduling systems. *Journal of Embedded Computing*, 1(2), 2004.

[8] G. Lipari, P. Gai, M. Trimarchi, G. Guidi, and P. Ancilotti. A hierarchical framework for component-based real-time systems. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 116, January 2005.

[9] S. Mazzini, M. D'Alessandro, M. Di Natale, G. Lipari, and T. Vardanega. Issues in mapping HRT-HOOD to UML. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, Porto, Portugal, July 2003.

[10] J. C. Palencia and M. González Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 26–37, Madrid, Spain, Dec. 1998.

[11] O. Redell and M. Sanfridson. Exact best-case response time analysis of fixed priority scheduled tasks. In *Proceeding of 14th Euromicro Conference on Real-Time Systems*, pages 165–172, Vienna, June 2002.

[12] I. Shih and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the 24th Real-Time Systems Symposium*, pages 2–13, Cancun, Mexico, Dec. 2003.

[13] A. Srinivasan and J. H. Anderson. Optimal rate-based scheduling on multiprocessors. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 189–198, Montreal, Canada, 2002.

[14] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Journal of Real Time Systems*, 6(2):133–151, Mar 1994.

[15] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing & Microprogramming*, 50(2–3):117–134, Apr. 1994.