Proc. of the 22nd Euromicro Conference on Real-Time Systems (ECRTS 10), Brussels, Belgium, July 6-9, 2010.

Partitioning parallel applications on multiprocessor reservations

Giorgio Buttazzo, Enrico Bini, Yifan Wu
Scuola Superiore Sant'Anna, Pisa, Italy
Email: {g.buttazzo,e.bini,y.wu}@sssup.it

***Abstract*—A full exploitation of the computational power available in a multi-core platform requires the software to be specified in terms of parallel execution flows. At the same time, modern embedded systems often consist of more parallel applications with timing requirements, concurrently executing on the same platform and sharing common resources. To prevent reciprocal interference among critical activities, a resource reservation mechanism is highly desired in the kernel to achieve temporal isolation.**

In this paper, we propose a general methodology for partitioning the total computing power available on a multi-core platform into a set of virtual processors, which provide a powerful abstraction to allocate applications independently of the physical platform. The application, described as a set of tasks with precedence relations expressed by a directed acyclic graph, is automatically partitioned into a set of subgraphs that are selected to minimize either the overall bandwidth consumption or the fragmentation of the partition (expressed by the so-called "λ-factor" in uniform multiprocessor scheduling).

I. Introduction

Multi-core architectures provide an efficient solution to the problem of increasing the processing speed with a contained power dissipation. However, analyzing multi-core systems is not trivial, and the research community is still working to produce new theoretical results or extend the well established theory for uniprocessor systems developed in the last 30 years. Also, fully exploiting the computational power available in a multi-core platform requires new programming paradigms, which should allow expressing the intrinsic parallel structure of the applications in order to optimize the allocation of parallel execution flows to different cores.

Moreover, the complexity of modern embedded systems is growing continuously, and the software is often structured in a number of concurrent applications, each consisting of a set of tasks with various characteristics and constraints, and sharing the same resources. In such a scenario, isolating the temporal behavior of real-time applications is crucial to prevent a reciprocal interference among critical activities.

Temporal isolation can be achieved through a *Resource Reservation* technique [1], [2], according to which the CPU processing capacity can be partitioned into a set of reservations, each equivalent to a virtual processor with reduced speed. In particular, a reservation is a couple (Q_k, P_k) indicating that at most Q_k units of time are available every period P_k. This means that the virtual processor has an equivalent bandwidth $\alpha_k = Q_k/P_k$. The main advantage of this approach is that an application allocated to a virtual machine can be guaranteed in "isolation" (i.e., independently of the other tasks in the system) only based on its timing requirements and on the amount of allocated bandwidth. In this way, overruns occurring in a task do not affect the temporal behavior of the other tasks.

When moving to multiprocessor systems, however, the meaning of reservations has to be revisited, and the research community just started to address this issue. The most natural abstraction of a multi-core platform is probably the uniform multiprocessor model proposed by Funk, Goossens and Baruah [3], where a collection of sequential machines is abstracted by their speeds. In this paper, the authors also showed that a set of tasks scheduled by global EDF (with migrations) and requiring an overall bandwidth of 120% has higher chances to be successfully scheduled upon two virtual processors with bandwidth 100% and 20%, rather than on other two with the same bandwidth of 60%. However, when no task migration is allowed, packing the bandwidth into full reservations is not always the best approach. In fact, consider a periodic application Γ consisting of 5 tasks with computation times 1, 1, 5, 6, 6 and period equal to 10 (deadline = period). In this case, the bandwidth required by the application is $U_\Gamma = 190\%$, and a feasible schedule can be found using 3 reservations, equal to 80%, 60% and 50%. However, no feasible solution exists if the bandwidth is provided by two reservations equal to 100% and 90%.

Otero et al. [4] applied the resource reservation paradigm to interrelated resources (processor cycles, cache space, and memory access cycles) to achieve robust, flexible and cost-effective consumer products.

Shin et al. [5] proposed a multiprocessor periodic resource model to describe the computational power supplied by a parallel machine. In their work, a resource is modeled using three parameters (P, Q, m), meaning that an overall budget Q is provided by at most m processors every period P.

Leontyev and Anderson [6] proposed a multiprocessor scheduling scheme for supporting hierarchical reservations (containers) that encapsulate hard and soft sporadic real-time tasks. Recently, Bini et al. [7] proposed to abstract a set of m virtual processors by the set of the m supply functions [8], [9], [10] of each virtual processors. In this paper we borrow such an abstraction of a virtual multi-core platform. In all these works, however, the application is modeled as a collection of sporadic tasks, and no precedence

This work has been partially supported by the ACTORS European project under contract 216586.

relations are taken into account.

A more accurate task model (generalized multiframe task) considering conditional execution flows, expressed by a Directed Acyclic Graph (DAG), has been proposed by Baruah et al. [11]. However, multiple branches outgoing from a node denote alternative execution flows rather than parallel computations.

The problem of managing real-time tasks with precedence relations was addressed by Chetto et al. [12], who proposed a general methodology for assigning proper activation times and deadlines to each task in order to convert a precedence graph into timing constraints, with the objective of guaranteeing the schedulability under EDF. Their algorithm, however, is only valid for uniprocessor systems and does not consider the possibility of having parallel computations.

Partitioning and scheduling tasks with precedence constraints onto a multiprocessor system has been shown to be NP-Complete in general [13], and various heuristic algorithms have been proposed in the literature to reduce the complexity [14], [15], [16], but their objective is to minimize the total completion time of the task set, rather than guaranteeing timing constraints under temporal isolation. One category of such algorithms, called List scheduling [15], [14], is based on proper priority assignments to meet the application constraints. Another technique, called Critical Path Heuristics [13], [16], was developed to deal with non-negligible communication delays between tasks. The idea is to assigns weights to nodes to reflect their resource usage and to edges to reflect the cost of inter-processor communication, and then shorten the length of the Critical Path of a DAG by reducing the communication between tasks within a cluster.

Collette et al. [17] proposed a model to express the parallelism of a code by characterizing all possible durations a computation would take on different number of processors. Schedulability is checked under global EDF, but no precedence relations are considered in the analysis.

Lee and Messerschmitt [18] developed a method to statically schedule synchronous data flow programs, on single or multiple processors. Precedence relations are considered in the model, but no deadline constraints are taken into account and temporal protection is not addressed.

Jayachandran and Abdelzaher [19] presented an elegant and effective algebra for composing the delay of applications modeled by DAGs and scheduled on distributed systems. However, they did not provide temporal isolation among applications.

Fisher and Baruah [20] derived near-optimal sufficient tests for determining whether a given collection of jobs with precedence constraints can feasibly meet all deadlines upon a specified multiprocessor platform under global EDF scheduling, so partitioning issues and resource reservations are not addressed.

Contribution of this work: In this paper, we propose a method for allocating a parallel real-time application, described as a set of tasks with time and precedence constraints, on a multi-core platform. To achieve modularity and simplify portability of applications on different multi-core platforms, we abstract the virtual platform by the Multi Supply Function (MSF) [7]. The advantage of using the virtual platform MSF is that, if the hardware platform is replaced with another one with a different number of cores, the set of reservations does not need to be changed, and only the server mapping to physical processors has to be done. Also, to be independent of a particular reservation algorithm, a virtual processor reservation is expressed by a bounded-delay time partition, denoted by the pair (α, Δ), where α is the allocated bandwidth and Δ is the maximum service delay. This method, originally proposed by Mok et al. [21], is general enough to express several types of resource reservation servers.

To better exploit the existing parallelism available in the computing platform, the application precedence graph is partitioned into a set of flows, each consisting of a subset of tasks to be sequentially executed on a virtual processor. For each flow, we determine its computational requirements and compute the minimum server bandwidth needed for executing it. Since the bandwidth requirements depend on the specific partition, the proposed method can be used to identify the partition that minimizes a given cost function (e.g., the overall bandwidth consumption or the application fragmentation).

Organization of the paper: The rest of the paper is organized as follows. Section II presents the system model, the terminology and the notation used throughout the paper, and recalls some background concepts. Section III describes the proposed method for selecting the optimal reservation parameters and the algorithm for partitioning the application into flows. Section IV illustrates some experimental results to validate the proposed approach. Finally, Section V states our conclusions and possible extensions for a future work.

II. System model and background

A real-time application is modeled as a set of tasks with given precedence constraints, specified as a Directed Acyclic Graph (DAG). Note that the DAG represents a description of the application considering the maximum level of parallelism. This means that each task represents a sequential activity to be executed on a single core. Tasks can be preempted at any time and do not call blocking primitives during their execution.

A. Terminology and notation

First, to shorten the expressions, we may denote $\max\{0, x\}$ as $(x)_0$. Moreover, throughout the paper we adopt the following terminology.

Application Γ: it is a set of n tasks with given precedence relations expressed by a Directed Acyclic Graph (DAG). The application is sporadic, meaning that it is cyclically activated with a minimum inter-arrival time T (also referred to as period) and must complete within a

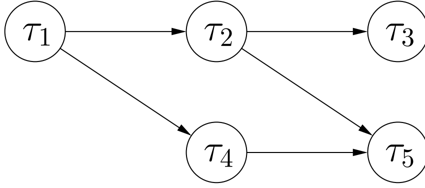

Figure 1. A sample application represented with a DAG.

given relative deadline D, which can be less than or equal to T. This allows asserting that only one instance of the application is running at any time.

Task τ_i: it is a portion of code that cannot be parallelized and must be executed sequentially. τ_i can be preempted at any time and is characterized by a known worst-case execution time $C_i > 0$. τ_i is also assigned a deadline d_i and an activation time a_i relative to the activation of the first task of the application. The assignment of deadlines and activation times is investigated in Section III-A. Tasks are scheduled by EDF.

Precedence relation $\mathcal{R}$: it is formally defined as a partial ordering $\mathcal{R} \subseteq \Gamma \times \Gamma$. Notation $\tau_i \prec \tau_j$ denotes that τ_i is a *predecessor* of τ_j, meaning that τ_j cannot start executing before the completion of τ_i. Notation $\tau_i \to \tau_j$ denotes that τ_i is an *immediate predecessor* of τ_j, meaning that $\tau_i \prec \tau_j$ and

$$\tau_i \prec \tau_k \prec \tau_j \Rightarrow (\tau_k = \tau_i \text{ or } \tau_k = \tau_j).$$

Figure 1 illustrates an example of DAG for an application consisting of five tasks, with execution times:

$$C_1 = 4, C_2 = 1, C_3 = 5, C_4 = 2, C_5 = 3.$$

The entire application starts at time $t = 0$ and is periodically activated with a period $T = 20$. We consider a relative deadline D equal to the period.

In addition, we define the following notation:

Path P: it is any subset of tasks $P \subseteq \Gamma$ totally ordered according to $\mathcal{R}$; i.e., $\forall \tau_i, \tau_j \in P$ either $\tau_i \prec \tau_j$ or $\tau_j \prec \tau_i$.

Execution time function $C(\cdot)$: it is a function $C : \mathcal{P}(\Gamma) \to \mathbb{R}$ that, applied to any subset A of Γ, returns the total execution time of the tasks in A:

$$\forall A \subseteq \Gamma \quad C(A) \stackrel{\text{def}}{=} \sum_{\tau_i \in A} C_i.$$

Sequential Execution Time C^s: it is the minimum time needed to complete the application on a uniprocessor, by serializing all tasks in the DAG. It is equal to the sum of all tasks computation times:

$$C^s \stackrel{\text{def}}{=} C(\Gamma).$$

For the application illustrated in Figure 1, we have $C^s = 15$.

Parallel Execution Time C^p: it is the minimum time needed to complete the application on a parallel architecture with an infinite number of cores. It is equal to

$$C^p \stackrel{\text{def}}{=} \max_{P \text{ is a path}} C(P). \qquad (1)$$

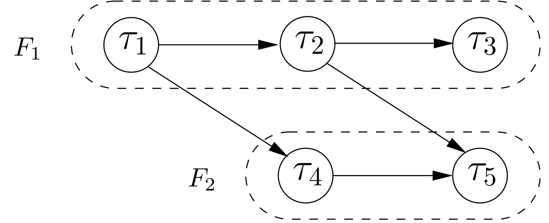

Figure 2. Parallel flows in which the application can be divided.

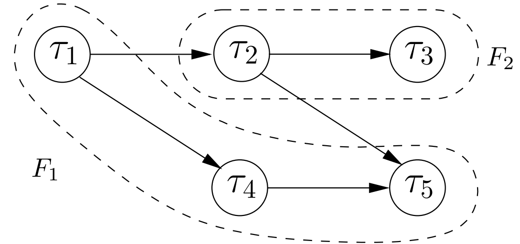

Figure 3. An alternate parallel flow selection.

Notice that the application relative deadline cannot be less than C^p, otherwise it is missed even on an infinite number of cores. For the application in Figure 1, we have $C^p = 10$.

Critical path (CP): it is a path P having $C(P) = C^p$.

Virtual processor VP_k: it is an abstraction of a sequential machine achieved through a resource reservation mechanism characterized by a bandwidth $\alpha_k \leq 1$ and a maximum service delay $\Delta_k \geq 0$.

Flow F_k: it is a subset of tasks $F_k \subseteq \Gamma$ allocated on virtual processor VP_k, which is dedicated to the execution of tasks in F_k only. Γ is partitioned into m flows.

Flow computation time C_k^F: it is the cumulative computation time of the tasks in flow F_k:

$$C_k^F \stackrel{\text{def}}{=} C(F_k).$$

Dividing an application into parallel flows allows several options, from the extreme case of defining a single flow for the entire application (where no parallelism is exploited/necessary and all tasks are sequentially executed on a single core) to the case of having a flow per task (maximum parallelism). The way in which flows are defined may affect the total bandwidth required to execute the application. Hence, we now address the problem of finding the best partition of flows that minimizes the total bandwidth requirements.

Intuitively, grouping tasks into large flows improves schedulability, as long as each flow has a bandwidth less than or equal to one. To better explain each step of the process, we consider a reference application consisting of five tasks, previously illustrated in Figure 1. For this example, we divide the application in two flows, as illustrated in Figure 2. Notice that there can be several ways for selecting flows in the same application. An alternative solution is shown in Figure 3.

B. Demand Bound Function

Since EDF is used as a scheduler, here we recall the concept of demand bound function that is used to estimate the amount of required computational resource. The processor demand of a task τ_i that has activation time a_i, computation time C_i, period T_i, and relative deadline d_i, in any interval $[t_1, t_2]$ is defined to be the amount of processing time $g_i(t_1, t_2)$ requested by those instances of τ_i activated in $[t_1, t_2]$ that must be completed in $[t_1, t_2]$. That is [22],

$$g_i(t_1, t_2) \stackrel{\text{def}}{=} \left(\left\lfloor \frac{t_2 - a_i - d_i}{T_i} \right\rfloor - \left\lceil \frac{t_1 - a_i}{T_i} \right\rceil + 1 \right)_0 C_i.$$

The overall demand bound function of a subset of tasks $A \subseteq \Gamma$ is

$$h(A, t_1, t_2) \stackrel{\text{def}}{=} \sum_{\tau_i \in A} g_i(t_1, t_2)$$

where we made it depend on the beginning and the length of the interval.

As suggested by Rahni et al. [23], we can use a more compact formulation of the demand bound function that depends only on the length t of the time interval $[t_1, t_1 + t]$:

$$\mathsf{dbf}(A, t) \stackrel{\text{def}}{=} \max_{t_1} h(A, t_1, t_1 + t). \tag{2}$$

C. The (α, Δ) server

Mok et al. [21] introduced the "bounded delay partition" to describe a reservation by the bandwidth α and the delay Δ. The bandwidth α measures the amount of resource that is assigned to the demanding application, whereas Δ represents the worst-case service delay.

Before introducing the α and Δ parameters, it is necessary to recall the concept of supply function [9], [10], that represents the minimum amount of time that a generic virtual processor can provide in a given interval of time.

Definition 1 (Def. 9 in [21], Th. 1 in [9], Eq. (6) in [24]): Given a virtual processor VP_k, its *supply function* $Z_k(t)$ is the minimum amount of time provided by the reservation in every time interval of length $t \geq 0$.

The supply function can be defined for many kinds of reservations, as static time partitions [21], [8], periodic servers [9], [10], or periodic servers with arbitrary deadline [24].

Given the supply function, the bandwidth α and the delay Δ can be formally defined as follows.

Definition 2 (compare Def. 5 in [21]): Given VP_k with supply function Z_k, the *bandwidth* α_k of the virtual processor is defined as

$$\alpha_k \stackrel{\text{def}}{=} \lim_{t \to \infty} \frac{Z_k(t)}{t}. \tag{3}$$

The Δ parameter provides a measure of the responsiveness, as proposed by Mok et al. [21].

Definition 3 (compare Def. 14 in [21]): Given VP_k with supply function Z_k and bandwidth α_k, the *delay* Δ_k of the virtual processor is defined as

$$\Delta_k \stackrel{\text{def}}{=} \sup_{t \geq 0} \left\{ t - \frac{Z_k(t)}{\alpha_k} \right\}. \tag{4}$$

If the (α, Δ) server is implemented through a periodic server [9], [10] that allocates a budget Q_k every period P_k, we have a bandwidth $\alpha_k = Q_k/P_k$ and a delay $\Delta_k = 2(P_k - Q_k)$. In practice, however, a portion of the processor bandwidth is wasted to perform context switches every time a server is executed. If σ is the run-time overhead required for a context switch, and P_k is the server period, the effective server bandwidth can be computed as:

$$B_k = \alpha_k + \frac{\sigma}{P_k}.$$

Expressing P_k as a function of α_k and Δ_k we have

$$P_k = \frac{\Delta_k}{2(1 - \alpha_k)}.$$

Hence,

$$B_k = \alpha_k + 2\sigma \frac{1 - \alpha_k}{\Delta_k}. \tag{5}$$

From previous results [10], we can state that a subset A is schedulable on the virtual processor characterized by bandwidth α and delay Δ, if and only if:

$$\forall t \geq 0 \quad \mathsf{dbf}(A, t) \leq \alpha(t - \Delta)_0. \tag{6}$$

III. Partitioning an application into flows

This section describes the method proposed in this paper to determine the optimal partition of an application into flows. A sample partition is depicted in Figure 4.

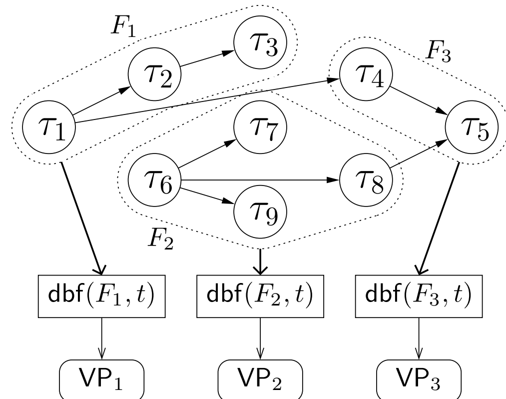

Figure 4. A sample partition into three flows.

The possible partitions into flows are explored through a branch and bound search algorithm, whose details are given later in Section III-C.

For a given partition (i.e., selection of flows), we first transform precedence relations into timing constraints by assigning suitable deadlines and activation times to each task, as illustrated in Section III-A.

Once deadlines and activations are assigned, the overall computational requirement of each flow F_k is evaluated

for all (nodes without successors) set $D_i = D$;
while (there exist nodes not set) {
  select a task τ_k with all successors modified;
  set $d_k = \min_{j:\tau_k \to \tau_j} (d_j - C_j)$;
}

Figure 5. The deadline assignment algorithm.

through its demand bound function and the parameters of the corresponding virtual processor VP_k are computed, as explained in Section III-B.

Then, if the objective is to minimize the total bandwidth, the overall bandwidth required by the entire partition is computed by summing the bandwidths computed for each flow using Equation (5) and, finally, the partition with the minimum bandwidth is determined as a result of the branch and bound search algorithm. A different metrics is also presented in Section III-C to minimize the fragmentation of the application.

A. Assigning deadlines and activations

Given a partition $\{F_1, \ldots, F_m\}$ of the application into m flows, activation times a_i and the deadlines d_i are assigned to all tasks to meet precedence relations and timing constraints. The assignment is performed according to a method originally proposed by Chetto-Silly-Bouchentouf [12], adapted to work on multi-core systems and slightly modified to reduce the bandwidth requirements. The algorithm starts by assigning the application deadline D to all tasks without successors. Then, the algorithm proceeds by assigning the deadlines to a task τ_i for which all successors have been considered. The deadline assigned to such a task is

$$d_i = \min_{j:\tau_i \to \tau_j} (d_j - C_j) \tag{7}$$

The pseudo-code of the deadline assignment algorithm is illustrated in Figure 5.

For the application shown in Figure 1, considering that the overall deadline is $D = T = 20$, by applying the transformation algorithm, we get:

$$\begin{aligned}
&d_3 = 20 \\
&d_5 = 20 \\
&d_2 = \min(d_3 - C_3, d_5 - C_5) = \min(15, 17) = 15 \\
&d_4 = d_5 - C_5 = 17 \\
&d_1 = \min(d_2 - C_2, d_4 - C_4) = \min(14, 15) = 14.
\end{aligned}$$

Activation times are set in a similar fashion, but we slightly modified the Chetto-Silly-Bouchentouf's algorithm to take into account that different flows can potentially execute in parallel on different cores. Clearly, τ_i cannot be activated before all its predecessors have finished.

Let τ_j be a predecessor of τ_i and let F_k be the flow τ_i belongs to. If $\tau_j \in F_k$, then the precedence constraint is already enforced by the deadline assignment given in Eq. (7).

for all (nodes without predecessors) set $a_i = 0$;
while (there exist nodes not set) {
  select a task τ_k with all predecessors modified;
  set $a_i = \max\{a_i^{\text{prec}}, d_i^{\text{prec}}\}$
}

Figure 6. The activation assignment algorithm.

Hence, it is sufficient to make sure that τ_i is not activated earlier than τ_j. In general, we must ensure that

$$a_i \geq \max_{\tau_j \to \tau_i, \tau_j \in F_k} \{a_j\} \stackrel{\text{def}}{=} a_i^{\text{prec}}. \tag{8}$$

On the other hand, if $\tau_j \notin F_k$, we cannot assume that τ_j will be allocated on the same physical core as τ_i, thus we do not know its precise finishing time. Hence, τ_i cannot be activated before τ_j deadline, that is

$$a_i \geq \max_{\tau_j \to \tau_i, \tau_j \notin F_k} \{d_j\} \stackrel{\text{def}}{=} d_i^{\text{prec}}. \tag{9}$$

In general, a_i must satisfy both (8) and (9). Moreover a_i should be as early as possible so that the resulting demand bound function is minimized [22]. Hence, we set

$$a_i = \max\{a_i^{\text{prec}}, d_i^{\text{prec}}\}. \tag{10}$$

Notice that, since d_i^{prec} depends on tasks belonging to other flows, it can be $a_i^{\text{prec}} > d_i^{\text{prec}}$.

The algorithm starts by assigning activation times to root nodes, i.e., tasks without predecessors. For such tasks, the activation time is set equal to the application activation time that we can assume to be zero, without loss of generality. Then, the algorithm proceeds by assigning activation times to a task for which all predecessors have been considered. Figure 6 illustrates the pseudo-code of the algorithm.

Indeed, the transformation algorithm proposed by Chetto, Silly, and Bouchentouf was designed to guarantee the precedence constraints, regardless of the processor demand. In fact it assigns deadlines as late as possible. However activations may coincide with some deadline as well. If an activation is too close to the corresponding deadline, then the demand bound function can become very large. To address this issue, in this work we propose an alternative deadline assignment that reduces the processor demand of the flow by distributing tasks deadlines more uniformly along the time line. If C^p is the computation time of a critical path and U^p is defined as

$$U^p = \frac{C^p}{D}$$

we propose to assign task deadlines as follows:

$$d_i = \min_{j:\tau_i \to \tau_j} (d_j - C_j / U^p) \tag{11}$$

instead of according to Eq. (7).

Experimental results reported in Section IV show that the modified assignment (referred to as *Chetto**) is able to achieve better performance with respect to the classical

Chetto assignment, especially for applications with complex precedence relations.

The following lemma shows that such a deadline assignment is sound, in the sense that all relative deadlines are greater than the cumulative computation times of the preceding tasks in a path.

Lemma 1: If each task τ_i of a path P is assigned a relative deadline

$$d_i = \min_{j:\tau_i \rightarrow \tau_j} (d_j - C_j/U^p)$$

where $U^p = C^p/D$, then it is guaranteed that all the tasks in P have relative deadlines greater than the cumulative execution time of the preceding tasks, that is

$$d_i \geq \sum_{\tau_k \in P, \tau_k \prec \tau_i} C_k.$$

Proof: Given any node τ_i, let $\tau_{i+1}, \tau_{i+2}, \ldots, \tau_L$ be the sequence of successors of τ_i such that τ_L is a leaf node (hence $d_L = D$) and

$$\forall j = i, \ldots, L-1 \quad d_j = d_{j+1} - C_{j+1}/U^p.$$

Then we have:

$$d_i = d_{i+1} - \frac{C_{i+1}}{U^p} = D - \frac{\sum_{j=i+1}^{L} C_j}{U^p}.$$

If P is a path including $\tau_i, \tau_{i+1}, \ldots, \tau_L$, we can write:

$$d_i = D - \frac{C(P) - \sum_{j=1}^{i} C_j}{U^p} = D - \frac{C(P)}{U^p} + \frac{\sum_{j=1}^{i} C_j}{U^p}$$

and since $U^p = C^p/D$ we have

$$d_i = D - \frac{C(P)}{C^p} D + \frac{\sum_{j=1}^{i} C_j}{U^p}.$$

Since $C(P) \leq C^p$ for any P, and $C^p \leq D$, we have:

$$d_i \geq \frac{\sum_{j=1}^{i} C_j}{U^p} \geq \sum_{j=1}^{i} C_j.$$

Thus, the lemma follows. ■

For the application shown in Figure 1, we have that:

$$\begin{aligned} a_1 &= 0 \\ D &= T = 20 \\ C^p &= 10 \\ C^s &= 15 \\ U^p &= \frac{C^p}{D} = 0.5 \end{aligned}$$

Hence, the proposed transformation algorithm (Eq. (11)) produces the following deadline assignment:

$$\begin{aligned} d_3 &= 20 \\ d_5 &= 20 \\ d_2 &= \min(20 - 5/0.5, 20 - 3/0.5) = \min(10, 14) = 10 \\ d_4 &= 20 - 3/0.5 = 14 \\ d_1 &= \min(10 - 1/0.5, 14 - 2/0.5) = \min(8, 10) = 8. \end{aligned}$$

If, for example, we select the flows $F_1 = \{\tau_1, \tau_2, \tau_3\}$ and $F_2 = \{\tau_4, \tau_5\}$, the activation times result to be:

$$\begin{aligned} a_1 &= 0 \\ a_2 &= 0 \\ a_3 &= 0 \\ a_4 &= d_1 = 8 \\ a_5 &= \max(a_4, d_2) = \max(8, 10) = 10 \end{aligned}$$

The demand bound functions of the two flows are derived according to Equation (2) and are illustrated in Figure 7 and Figure 8, respectively.

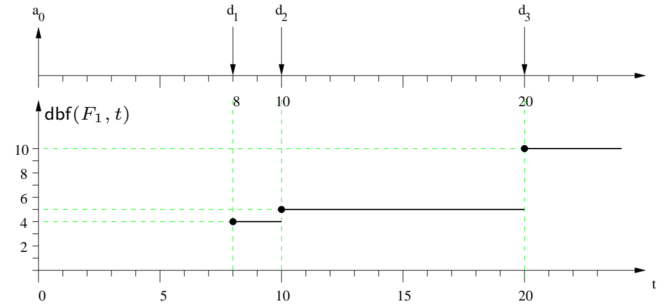

Figure 7. Demand bound function of flow F_1.

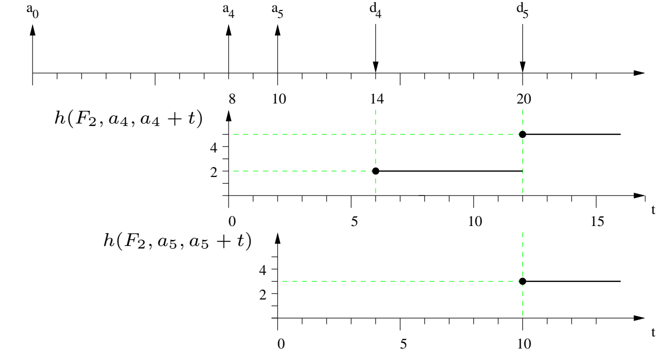

Figure 8. Demand bound function of flow F_2.

B. Bandwidth requirements for a flow

Once activation times and deadlines have been set for all tasks, each flow can be independently executed on different virtual processors under EDF, in isolation, ensuring that precedence constraints are met.

To determine the reservation parameters that guarantee the feasibility of the schedule, we need to characterize the computational requirement of each flow. By using the

demand bound function defined in Equation (2) we have that a flow F is schedulable on the virtual processor VP characterized by bandwidth α and delay Δ if and only if:

$$\forall t \geq 0 \quad \mathsf{dbf}(F, t) \leq \alpha(t - \Delta)_0. \tag{12}$$

Now the problem is to select the (α, Δ) parameters among all possible pairs that satisfy Eq. (12). We propose to select the pair that minimizes the bandwidth B used by the virtual processor, as given by Eq. (5), which accounts for the cost of the server overhead. Hence, the best (α, Δ) pair is the solution of the following minimization problem:

$$\begin{aligned} &\text{minimize} \quad \alpha + \varepsilon \frac{1-\alpha}{\Delta} \\ &\text{subject to} \quad \mathsf{dbf}(F, t) \leq \alpha(t - \Delta)_0, \quad \forall t \geq 0, \end{aligned} \tag{13}$$

with $\varepsilon = 2\sigma$.

This problem has been shown [25] to have a very efficient solution that exploits the convexity of the domain and the quasiconvexity of the cost function. Hence we adopt the solution proposed in [25].

C. The branch and bound algorithm

This section illustrates the algorithm used for selecting the best partition of the application into flows. Two different objectives have been considered in the optimization procedure.

As a first optimization goal, we considered minimizing the overall bandwidth requirement of the selected flows, that is

$$B = \sum_{k=1}^{m} B_k = \sum_{k=1}^{m} \left(\alpha_k + 2\sigma \frac{1-\alpha_k}{\Delta_k} \right). \tag{14}$$

Clearly, the number m of flows has to be determined as well.

As a second optimization goal, we considered minimizing the fragmentation of a partition, defined as

$$\beta = \max_{k=1,\ldots,m} \frac{\sum_{i=k}^{m} B_i}{B_k}, \tag{15}$$

where the bandwidths $B_1, \ldots, B_m$ are assumed to be ordered by non increasing values. The selection of this metric is inspired by the global EDF test on uniform multiprocessors [3]. In fact, in uniform multiprocessor scheduling, if $B_1 \geq B_2 \geq \ldots \geq B_m$ are the speeds of the processors, a platform with a low value of β has higher chance to schedule tasks due to the lower degree of fragmentation of the overall computing capacity[1].

To show the benefit of adopting the cost of Equation (15), let us consider a virtual platform with m identical processors, each providing $B_k = B/m$. While the cost according to Eq. (14) is B, hence independent of the number of virtual processors, the cost according to Eq. (15) is m. It follows that the minimization of β leads to the reduction of number of flows in which the application is partitioned. Nonetheless, the minimization of β also implicitly implies the selection of a partitioning with low overall bandwidth requirement B. In fact we have that

$$B = \sum_{i=1}^{m} B_i \leq \frac{\sum_{i=1}^{m} B_i}{B_1} \leq \max_{k=1,\ldots,m} \frac{\sum_{i=k}^{m} B_i}{B_k} = \beta.$$

Hence β is also an upper bound of the overall bandwidth B, and a minimization of β leads indirectly to the selection of a low value of B as well.

The search for the optimal flow partition is approached by using a branch and bound algorithm, which explores the possible partitions by generating a search tree as illustrated in Figure 9.

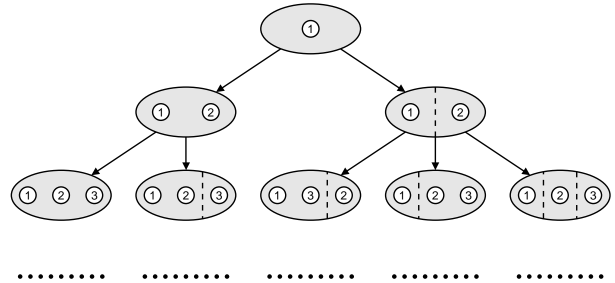

Figure 9. The search tree.

At the root level (level 1), task τ_1 is associated with flow F_1. At level 2, τ_2 is assigned either to the same flow F_1 (left branch) or to a newly created flow F_2 (right branch). In general, at each level i, task τ_i is assigned either to one of the existing flows, or to a new created flow. Hence, the depth of the tree is equal to the number n of tasks composing the application, whereas the number of leaves of the tree is equal to the number of all the possible partitions of a set of n members, given by the Bell Number b_n [26], recursively computed by

$$b_{n+1} = \sum_{k=0}^{n} \binom{n}{k} b_k = \sum_{k=0}^{n} \frac{n!}{k!\,(n-k)!}\, b_k. \tag{16}$$

To reduce the average complexity of the search, we use some pruning conditions to cut unfeasible and redundant branches for improving the run-time behavior of the algorithm.

We first observe that if, at some node, there is a flow F_k with bandwidth greater than one

$$B_k \geq \sum_{\tau_i \in F_k} \frac{C_i}{T} > 1 \tag{17}$$

then the schedule of the tasks in that flow is unfeasible, since

$$\sum_{\tau_i \in F_k} C_i > T \geq D. \tag{18}$$

Hence, whenever a node has a flow with bandwidth greater than one, we can prune the whole subtree, since no feasible partitioning can be found in the subtree. Moreover, the pruning efficiency can be further improved by allocating tasks by decreasing computation times, because this order

[1]Notice that in [3] the authors use $\lambda = \beta - 1$ to express the parallelism of the platform.

allows pruning a subtree satisfying Eq. (17) at the highest possible level.

The following lemma provides a lower bound on the number of flows in any feasible partition:

Lemma 2: In any feasible partitioning, the number of flows satisfies

$$m \geq \left\lceil \frac{C^s}{D} \right\rceil . \quad (19)$$

Proof: In any feasible partitioning $\{F_1, \ldots, F_m\}$, we have

$$\frac{C(F_k)}{D} \leq 1. \quad (20)$$

Adding equations (20) for all the flows, we have

$$\frac{\sum_k C(F_k)}{D} = \frac{C^s}{D} \leq m$$

And since m is integer,

$$m \geq \left\lceil \frac{C^s}{D} \right\rceil .$$

■

Nonetheless, much of the complexity of the algorithm lies in the horizontal expansion of the tree: in fact, the search tree keeps adding possible new flows (at the rightmost branch) even when the number of flows is higher than the parallelism that can be possibly exploited by the application. Hence, we prune a subtree when the number of flows exceeds a given bound m_{max}. A tight value of m_{max} is not easy to find, hence we adopted the following heuristic value:

$$m_{\mathsf{max}} = \left\lceil \delta \frac{C^s}{D} \right\rceil \quad (21)$$

where $\delta \geq 1$ is a parameter for tuning the size of the search tree. A value of δ close to one allows a significant improvement in terms of execution time, but at the price of losing optimality. Larger values of δ permit reaching optimality with reasonable execution times. As illustrated in the next section, our simulation results show that the optimal solution is often achieved with $\delta \leq 2$.

IV. Experimental results

To illustrate the effectiveness of the proposed search algorithm, in this section we present a number of experiments aimed at comparing the performance of the produced solution (in terms of number of flows and required bandwidth) and the efficacy of the pruning rules (in terms of reducing the number of steps).

In a first experiment, we considered the application shown in Figure 4, consisting of $n = 9$ tasks with computation times $C_1 = 2$, $C_2 = 3$, $C_3 = 5$, $C_4 = 3$, $C_5 = 4$, $C_6 = 3$, $C_7 = 6$, $C_8 = 5$, and $C_9 = 6$. From the DAG of the application, it results that the sequential execution time is $C^s = 37$ and the parallel execution time is $C^p = 12$, corresponding to the critical path $P = \{\tau_6, \tau_8, \tau_5\}$. Notice that the ratio $\pi = C^s/C^p$ provides an indication of the maximum level of parallelism of the application. In this example, we have $\pi \simeq 3.08$. Clearly, when the application deadline D is less than C^p, the schedule is infeasible on any number of cores, whereas when $D = C^p = 12$, the number of cores cannot be less than 4 (see Lemma 2).

Figure 10 reports the bandwidth B required by the optimal partition (including the context switch overhead σ), as a function of the application deadline D (ranging from C^p to C^s), using the first optimization goal expressed by Eq. (14). The figure also reports the minimum theoretical bound C^s/D (without overhead) and the worst-case bandwidth obtained by selecting one flow per task. Notice that the solution found by the algorithm is always very close to the ideal one and significantly better than the worst-case curve.

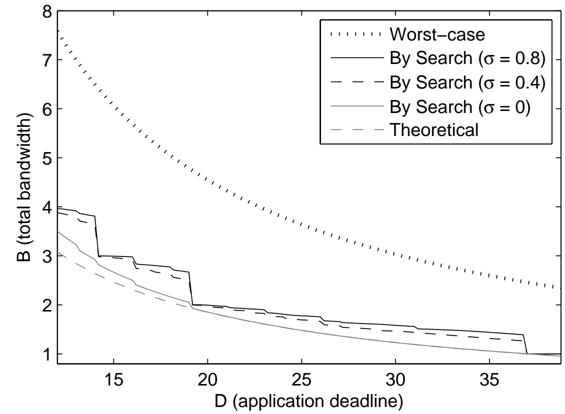

Figure 10. Total bandwidth as a function of the application deadline.

Considering the second optimization goal, expressed by the cost function reported by Eq. (15), Figure 11 reports the optimal β achieved by the search algorithm, as a function of the application deadline, for different values of σ.

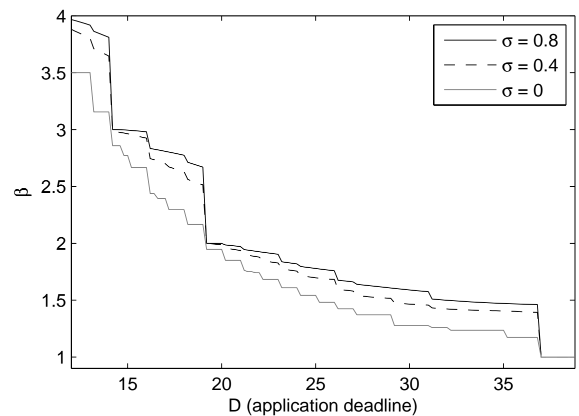

Figure 11. β as a function of the application deadline.

The difference between the bandwidth achieved by the second and the first optimization goal was also measured, but it was never found larger than 0.12. Hence, in the following experiments β was used as a performance metrics, since also

aimed at reducing fragmentation among different cores.

To test the run-time behavior of the search algorithm and the efficiency of the pruning rule, we ran another experiment with a fully parallel application (i.e., no precedence relations) with random computation times, generated with uniform distribution in [1,10]. The application deadline was set between C^p and C^s, with a value $D = (C^p + C^s)/2$. The run-time behavior of the algorithm was monitored by counting the number of steps for reaching a solution, as a function of the number of tasks, for different values of the pruning parameter δ. The results of this experiment are shown in Figure 12, which clearly shows that a considerable amount of steps are saved when small values of δ are used. It is worth mentioning that using a small value of δ results in negligible bandwidth loss. Intuitively, this can be justified by considering that a high number of flows often requires a high total B.

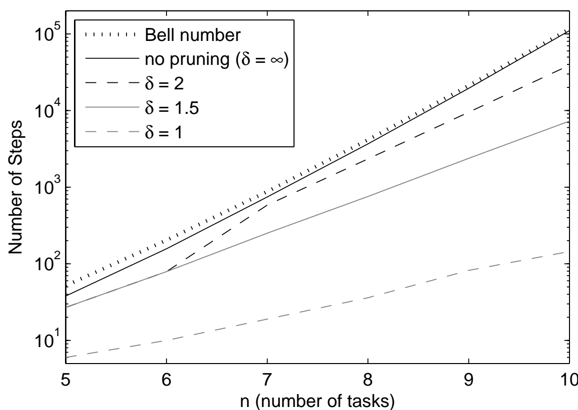

Figure 12. Run-time of the algorithm as a function of n.

To investigate the effectiveness of the proposed method for assigning deadlines (denoted as *Chetto**), other two tests were performed against the original Chetto's method (denoted as *Chetto*) and Simulated Annealing (denoted as *SA*). An application with 16 tasks was generated, with computation times uniformly distributed in [1,10]. The tasks were connected with 25 precedences, giving $C^p = 27.6$, $C^s = 87.7$ and $\pi \simeq 3.18$. The application deadline was set to 42.

The first test was aimed at monitoring β as a function of the complexity of the precedence graph, measured as the number of the precedence links. At each step of the simulation, a random precedence link in the application was dropped, excluding those in the critical path, which was kept during the whole simulation to keep U^p constant. Then, the optimal β was computed using the three deadline assignment methods. Each point on the graph, was computed as the average on 10 simulations (differing on the sequence of random precedence link deletions). The result of this first test are reported in Figure 13, which shows that β increases with the number of precedence links. However, the β obtained by *Chetto** is smaller than that achieved by *Chetto* and close to the value found by simulated annealing. Differences become more significant as the number of precedence links increases.

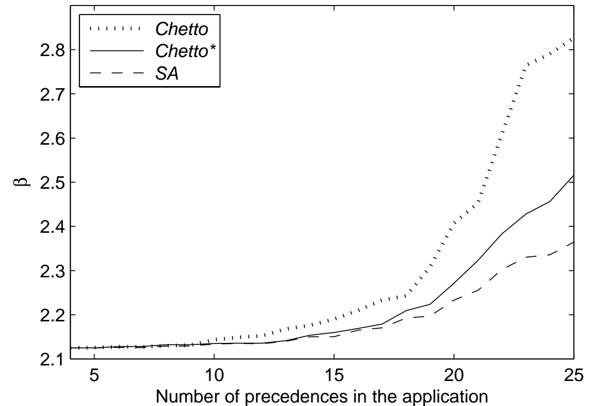

Figure 13. β as a function of precedence number.

The second test was aimed at evaluating the gap between *Chetto** and *SA*, as a function of the application deadline. All 25 precedence links were kept, and the application deadline was varied from 27.6 to 97.6 with steps of 2.5. The results reported in Figure 14 show that the difference of β obtained by *Chetto** and *SA* is quite small, meaning that the proposed deadline assignment method is close to the optimal deadline assignment, and can be confidently used in practice.

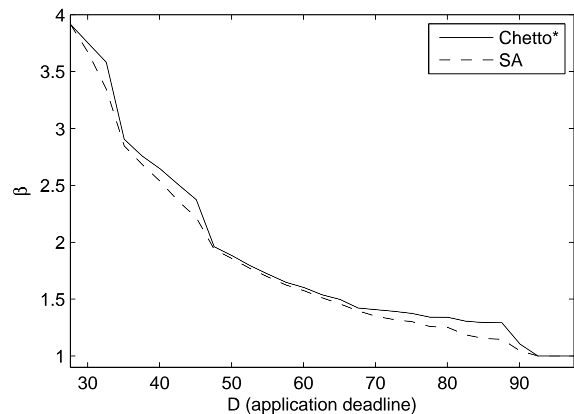

Figure 14. Comparison of *Chetto** and *SA*.

V. Conclusions

This paper presented a general methodology for allocating a parallel real-time application to a multi-core platform in a way that is independent of the number of physical cores available in the hardware architecture. Independence is achieved through the concept of virtual processor, which abstracts a resource reservation mechanism by means of two parameters, α (the bandwidth) and Δ (the maximum service delay).

The major contribution of this work was the development of an algorithm that automatically partitions the application into flows, in order to meet the specified timing constraints and minimize either the overall required bandwidth B or the fragmentation β. The computational requirements of each flow are derived through the processor demand criterion, after defining intermediate activation times and deadlines for each task, properly selected to satisfy precedence relations and timing constraints.

As a future work, we plan to extend the virtual processor allocation algorithm under high throughput requirements, achieved through pipelined executions, possibly using the methods in [19]. We also plan to integrate the proposed technique in a graphical tool for supporting the design of parallel real-time applications on multi-core platforms.

REFERENCES

[1] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves for multimedia operating systems," Carnegie Mellon University, Pittsburg, Tech. Rep. CMU-CS-93-157, May 1993.

[2] L. Abeni and G. Buttazzo, "Resource reservation in dynamic real-time systems," *Real-Time Systems*, vol. 27, no. 2, pp. 123–167, Jul. 2004.

[3] S. Funk, J. Goossens, and S. Baruah, "On-line scheduling on uniform multiprocessors," in *Proceedings of the* 22nd *IEEE Real-Time Systems Symposium*, London, United Kingdom, Dec. 2001, pp. 183–192.

[4] C. Otero Pérez, M. Rutten, L. Steffens, J. van Eijndhoven, and P. Stravers, "Resource reservations in shared-memory multiprocessor SoCs," in *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, B. S. P. Research, Ed. Netherlands: Springer, 2006, ch. 5, pp. 109–137.

[5] I. Shin, A. Easwaran, and I. Lee, "Hierarchical scheduling framework for virtual clustering multiprocessors," in *Proceedings of the* 20th *Euromicro Conference on Real-Time Systems*, Prague, Czech Republic, Jul. 2008, pp. 181–190.

[6] H. Leontyev and J. H. Anderson, "A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees," in *Proceedings of the* 20th *Euromicro Conference on Real-Time Systems*, Prague, Czech Republic, Jul. 2008, pp. 191–200.

[7] E. Bini, G. C. Buttazzo, and M. Bertogna, "The multy supply function abstraction for multiprocessors," in *Proceedings of the* 15th *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Beijing, China, Aug. 2009, pp. 294–302.

[8] X. Feng and A. K. Mok, "A model of hierarchical real-time virtual resources," in *Proceedings of the* 23rd *IEEE Real-Time Systems Symposium*, Austin, TX, U.S.A., Dec. 2002, pp. 26–35.

[9] G. Lipari and E. Bini, "Resource partitioning among real-time applications," in *Proceedings of the* 15th *Euromicro Conference on Real-Time Systems*, Porto, Portugal, Jul. 2003, pp. 151–158.

[10] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proceedings of the* 24th *Real-Time Systems Symposium*, Cancun, Mexico, Dec. 2003, pp. 2–13.

[11] S. K. Baruah, D. Chen, S. Gorinsky, and A. K. Mok, "Generalized multiframe tasks," *Real-Time Systems*, vol. 17, no. 1, pp. 5–22, Jul. 1999.

[12] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *Real-Time Systems*, vol. 2, no. 3, pp. 181–194, Sep. 1990.

[13] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Cambridge, MA, USA: MIT Press, 1989.

[14] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Communications of the ACM*, vol. 17, no. 12, pp. 685–690, 1974.

[15] H. El-Rewini and T. G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *Journal of Parallel and Distributed Computing*, vol. 9, no. 2, pp. 138–153, 1990.

[16] Y. kwong Kwok, I. Ahmad, and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, pp. 506–521, 1996.

[17] S. Collette, L. Cucu, and J. Goossens, "Integrating job parallelism in real-time scheduling theory," *Information Processing Letters*, vol. 106, no. 5, pp. 180–187, May 2008.

[18] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. 36, no. 1, pp. 24–35, 1987.

[19] P. Jayachandran and T. Abdelzaher, "Delay composition algebra: A reduction-based schedulability algebra for distributed real-time systems," in *Proceedings of the* 29th *IEEE Real-Time Systems Symposium*, Barcelona, Spain, Dec. 2008, pp. 259–269.

[20] N. Fisher and S. Baruah, "The feasibility of general task systems with precedence constraints on multiprocessor platforms," *Real-Time Systems*, vol. 41, no. 1, pp. 1–26, 2009.

[21] A. K. Mok, X. Feng, and D. Chen, "Resource partition for real-time systems," in *Proceedings of the* 7th *IEEE Real-Time Technology and Applications Symposium*, Taipei, Taiwan, May 2001, pp. 75–84.

[22] S. K. Baruah, R. Howell, and L. Rosier, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real-Time Systems*, vol. 2, pp. 301–324, 1990.

[23] A. Rahni, E. Grolleau, and M. Richard, "Feasibility analysis of non-concrete real-time transactions with edf assignment priority," in *Proceedings of the* 16th *conference on Real-Time and Network Systems*, Rennes, France, Oct. 2008, pp. 109–117.

[24] A. Easwaran, M. Anand, and I. Lee, "Compositional analysis framework using EDP resource models," in *Proceedings of the* 28th *IEEE International Real-Time Systems Symposium*, Tucson, AZ, USA, 2007, pp. 129–138.

[25] E. Bini, G. Buttazzo, and Y. Wu, "Selecting the minimum consumed bandwidth of an EDF task set," in *Proceedings of the* 2nd *Workshop on Compositional Real-Time Systems*, Washington, DC, U.S.A., Dec. 2009.

[26] G. Rota, "The number of partitions of a set," *American Mathematical Monthly*, vol. 71, no. 5, pp. 498–504, 1964.