Partitioning Real-Time Applications Over Multicore Reservations

Giorgio Buttazzo, *Senior Member, IEEE*, Enrico Bini, *Member, IEEE*, and Yifan Wu, *Member, IEEE*

***Abstract*—A full exploitation of the computational power available in a multicore 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 abstracting the total computing power available on a multicore platform by a set of virtual processors, 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 required number of cores.

***Index Terms*—Multiprocessor, partition algorithm, real-time systems, resource reservation.**

I. Introduction

In EMBEDDED systems, optimizing the available resources while meeting a desired performance is a crucial design objective, which can have a significant impact on the overall system cost, in terms of money, energy, space, weight, etc., depending on the specific application domain. Moreover, the continuous increase of complexity and the higher performance requirements of the new products is pushing the industry to adopt multicore platforms in several domains, like consumer electronics, and automotive systems. As a consequence, the analysis of these platforms is receiving growing attention both in the research and industrial community [1].

Multicore architectures provide an efficient solution to the problem of increasing the processing speed with a contained power dissipation. In fact, increasing the operating frequency of a single processor would cause serious heating problems and a considerable power consumption. In a multicore platform, however, the way tasks are allocated to processors significantly affects the number of active cores required for running the application, hence, minimizing such a number is of crucial importance for different design objectives, such as guaranteeing the feasibility of the system under given performance requirements, applying energy-aware strategies by turning some cores off, or fitting large applications into platforms with reduced number of cores.

On the other hand, analyzing multicore 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 multicore 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 [2], [3], 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 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 applications in the system) only based on its timing requirements and on the amount of allocated bandwidth. In this way, overruns occurring in an application do not affect the temporal behavior of the other applications. Resource reservation is a powerful methodology that can be applied to isolate the behavior of real-time, as well as non real-time applications [4].

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 multicore platform is probably the uniform multiprocessor model proposed by Funk *et al.* [5], 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 the global Earliest Deadline First (EDF) algorithm (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 five tasks with computation times

Manuscript received July 03, 2010; revised September 28, 2010; accepted February 08, 2011. Date of publication March 24, 2011; date of current version May 06, 2011. This work was supported in part by the ACTORS European Project under Contract 216586. Paper no. TII-10-07-0180.

G. Buttazzo and E. Bini are with the Scuola Superiore Sant'Anna, Pisa 56127, Italy (e-mail: g.buttazzo@sssup.it; e.bini@sssup.it).

Y. Wu is with Hangzhou Dianzi University, Zhejiang 310018, China (e-mail: yfwu@hdu.edu.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TII.2011.2123902

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 three 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%.

Although several approaches have been proposed in the literature to partition real-time applications over a set of processing elements [6]–[8], they mainly concentrated on the problem of achieving a feasible schedule or minimizing the maximum response time among the tasks, but did not address the problem of minimizing the computational resources. With the rapid development of multicore embedded systems, minimizing the available resources is of crucial importance to save energy or keep chip temperature under control.

A. *Contribution of This Work*

This paper describes a method for partitioning a parallel real-time application into a set of sequential flows, with utilization less than one, that can be allocated to specific processors to minimize resource consumption (in terms of overall computational bandwidth or number of processors). An application is described as a set of tasks with time and precedence constraints and represented through a directed acyclic graph. To achieve modularity and simplify portability on different architectures, the partitioning process is performed on a virtual platform, which abstracts the physical platform using a set of uniprocessor reservations described through the Multisupply Function (MSF) model proposed by Bini *et al.* [9]. Reservations can then be implemented by an aperiodic server or any resource reservation mechanism.

The advantage of using a virtual platform is that, if the hardware platform is replaced with another one with a different number of cores, the partitioning process does not need to be changed and only the mapping of the reservations to physical processors has to be done. Moreover, 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.* [10], is general enough to express several types of resource reservation servers.

The work presented here considerably extends a previous paper [11] by adding several key contributions. Since the original branch and bound search algorithm is too complex to handle applications with more than 20 tasks, this paper proposes some heuristic algorithms with reduced complexity that can find suboptimal partitions of applications consisting of hundreds of tasks, hence making the approach more usable for practical purposes. Also, a detailed explanation has been added to illustrate how to solve the optimization problem for selecting the best alpha-delta parameters of a virtual processor. The pseudocode of the proposed algorithms is included for completeness and a set of new simulation experiments has been reported for evaluating the performance of the proposed heuristics with respect to the branch and bound search.

Since the proposed methodology is independent of the physical platform, all results presented in this paper apply to both multicore and multiprocessor systems. Finally note that, the described methodology can be fruitfully integrated within more complex design and verification tools, as the one presented in [12].

B. *Organization of the Paper*

The rest of this 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 presents some heuristics to achieve the same goal with reduced complexity. Section V illustrates some experimental results to validate the proposed approach. Section VI discusses some related work. Finally, Section VII 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. Notice that preventing the use of blocking primitives inside the task code does not necessarily mean preventing data sharing among the tasks. In fact, tasks can communicate through nonblocking mechanisms that use memory duplication to avoid blocking on critical sections. An example of such a mechanism is represented by the *Cyclic Asynchronous Buffer* (CAB) [13].

A. *Terminology and Notation*

First, to shorten the expressions, we may denote $\max\{0, x\}$ as $(x)_0$. Moreover, throughout this 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 interarrival time T (also referred to as period) and must complete within a 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. This means that if the application is activated at t the task τ_i must execute in $[t + a_i, t + d_i]$. 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

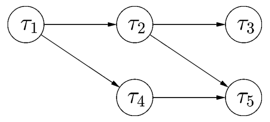

Fig. 1. A sample application represented with a DAG.

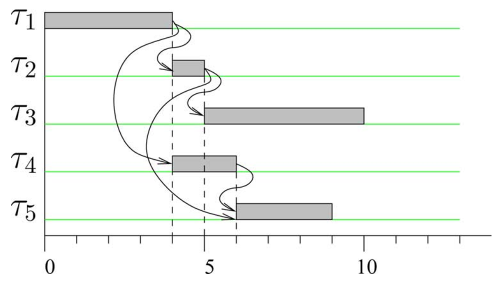

Fig. 2. Timeline representation.

the completion of τ_i. Notation $\tau_i \rightarrow \tau_j$ denotes that τ_i is an *immediate predecessor* of τ_j, meaning that $\tau_i \prec \tau_j$ and

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

Fig. 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.

To better illustrate the parallel execution of an application and identify the maximum number of required processors, we adopt a different description that visualizes the computation times of each task in the timeline, as in a Gantt chart. In such a diagram, denoted as the *timeline representation*, each task starts as soon as possible on the first available core, assuming as many cores as needed. For the application shown in Fig. 1, the timeline representation is illustrated in Fig. 2, where synchronization points coming from the precedence graph are represented by arrows.

An advantage of the timeline representation is that it clearly visualizes the intrinsic parallelism of the application, showing in each time slot the maximum number of cores needed to perform the required computation. This means that adding other cores will not reduce the overall response time, because the DAG already expresses the maximum level of parallelism.

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 $\tau_j \prec \tau_i$.

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

$$C^s \overset{\text{def}}{=} \sum_{\tau_i \in \Gamma} C_i.$$

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

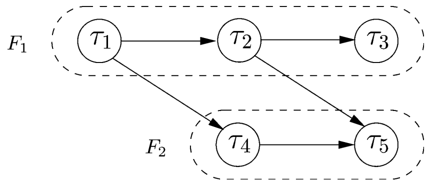

Fig. 3. Parallel flows in which the application can be divided.

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

$$C^p \overset{\text{def}}{=} \max_{P \text{ is a path}} \sum_{\tau_i \in P} C_i. \tag{1}$$

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 Fig. 1, we have $C^p = 10$.

Critical Path (CP): It is a path P having $\sum_{\tau_i \in P} C_i = C^p$. Notice that in the example of Fig. 1 the critical path is $P = \{\tau_1, \tau_2, \tau_3\}$.

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

$$C_k^F \overset{\text{def}}{=} \sum_{\tau_i \in F_k} C_i.$$

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 Fig. 1. For this example, we divide the application in two flows, as illustrated in Fig. 3. Notice that there can be several ways for selecting flows in the same application. An alternative solution is shown in Fig. 4.

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, 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

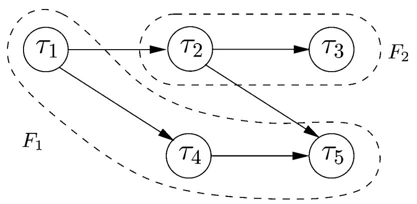

Fig. 4. An alternate parallel flow selection.

instances of τ_i activated in $[t_1, t_2]$ that must be completed in $[t_1, t_2]$. That is [14]

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

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.* [15], 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.* [10] 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 [16], [17], that represents the minimum amount of time that a generic virtual processor can provide in a given interval of time.

Definition 1 ([10, Def. 9], [16, Th. 1], [18, (6)]): 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 [10], [19], periodic servers [16], [17], or periodic servers with arbitrary deadline [18]. For example, for the simple case of a periodic reservation that allocates Q units of time every period P, we have [16], [17]:

$$Z(t) = \max\{0, t - P + Q - (k+1)(P-Q), kQ\} \tag{3}$$

with $k = \lfloor t - P + Q/P \rfloor$.

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

Definition 2 (Compare Def. 5 in [10]): Given VP_k with supply function Z_k, the *bandwidth* α_k of the virtual processor is defined as

$$\alpha_k \stackrel{\text{def}}{=} \lim_{t \to \infty} \frac{Z_k(t)}{t}. \tag{4}$$

The Δ parameter provides a measure of the responsiveness, as proposed by Mok *et al.* [10].

Definition 3 (Compare Def. 14 in [10]): Given VP_k with supply function Z_k and bandwidth α_k, the *delay* Δ_k of the virtual processor is defined as

$$\Delta_k \stackrel{\text{def}}{=} \sup_{t \geq 0} \left\{ t - \frac{Z_k(t)}{\alpha_k} \right\}. \tag{5}$$

Informally speaking, given a VP ν with bandwidth α_ν, the delay Δ_ν is the minimum horizontal displacement such that the line $\alpha_\nu(t - \Delta_\nu)$ is a lower bound of $Z_\nu(t)$.

Once the bandwidth and the delay are computed, the supply function of VP_k can be lower bounded as follows:

$$Z_k(t) \geq \alpha_k(t - \Delta_k)_0. \tag{6}$$

If the (α, Δ) server is implemented through a periodic server that allocates a budget Q_k every period P_k, then the bandwidth α_k is equal to Q_k/P_k and a delay $\Delta_k = 2(P_k - Q_k)$, as shown in [16] and [17]. 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 runtime 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

$$B_k = \alpha_k + 2\sigma \frac{1 - \alpha_k}{\Delta_k}. \tag{7}$$

From previous results [17], we can state that a subset A is schedulable on the virtual processor characterized by bandwidth α and delay Δ, if and only if

$$\forall t \geq 0 \quad \mathsf{dbf}(A,t) \leq \alpha(t - \Delta)_0. \tag{8}$$

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 Fig. 5.

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

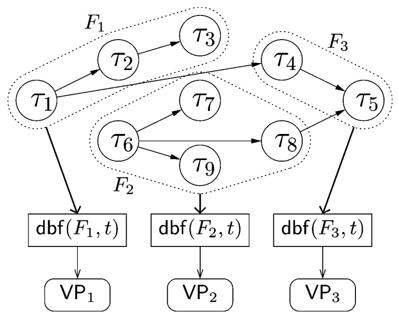

Fig. 5. A sample partition into three flows.

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

Fig. 6. The deadline assignment algorithm.

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 through its demand bound function and the parameters of the corresponding virtual processor VP$_k$ are computed, as explained in Section III-C.

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 (7) 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-D 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 *et al.* [20], adapted to work on multicore 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

$$d_i = \min_{j:\tau_i \to \tau_j} (d_j - C_j). \tag{9}$$

The pseudocode of the deadline assignment algorithm is illustrated in Fig. 6.

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

Fig. 7. The activation assignment algorithm.

For the application shown in Fig. 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 (9). Hence, it is sufficient to make sure that τ_i is not activated earlier than τ_j. In general, we must ensure that

$$a_i \geq a_i^{\text{prec}} \stackrel{\text{def}}{=} \max_{\tau_j \to \tau_i, \tau_j \in F_k} \{a_j\}. \tag{10}$$

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's deadline, that is

$$a_i \geq d_i^{\text{prec}} \stackrel{\text{def}}{=} \max_{\tau_j \to \tau_i, \tau_j \notin F_k} \{d_j\}. \tag{11}$$

In general, a_i must satisfy both (10) and (11). Moreover, a_i should be as early as possible so that the resulting demand bound function is minimized [14]. Hence, we set

$$a_i = \max\left\{a_i^{\text{prec}}, d_i^{\text{prec}}\right\}. \tag{12}$$

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. Fig. 7 illustrates the pseudocode of the algorithm.

Indeed, the transformation algorithm proposed by Chetto *et al.* 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, according to (12). 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 timeline. 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:

$$d_i = \min_{j:\tau_i \to \tau_j} \left(d_j - \frac{C_j}{U^p} \right) \tag{13}$$

instead of according to (9).

Experimental results reported in Section V 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 or equal to the cumulative computation time of the preceding tasks in a path.

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

$$d_i = \min_{j:\tau_i \to \tau_j} \left(d_j - \frac{C_j}{U^p} \right)$$

where $U^p = C^p/D$, then it is guaranteed that all the tasks in P have relative deadlines greater than or equal to 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

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

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

$$\begin{aligned} d_i &= D - \frac{\sum_{\tau_j \in P} C_j - \sum_{j=1}^{i} C_j}{U^p} \\ &= D - \frac{\sum_{\tau_j \in P} C_j}{U^p} + \frac{\sum_{j=1}^{i} C_j}{U^p} \end{aligned}$$

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

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

Since $\sum_{\tau_j \in P} C_j \leq C^p$ 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. ■

B. Example

This section presents an example to show how to derive the computational demand of a partition for the application illustrated in Fig. 1, characterized by the following parameters:

$$\begin{aligned} a_0 &= 0 \\ D &= T = 20 \\ C^p &= 10 \\ C^s &= 15 \\ U^p &= \frac{C^p}{D} = 0.5. \end{aligned}$$

Let us consider the partition depicted in Fig. 3, consisting of two flows: $F_1 = \{\tau_1, \tau_2, \tau_3\}$ and $F_2 = \{\tau_4, \tau_5\}$.

Applying the proposed deadline transformation algorithm [(13)], the following intermediate deadlines can be assigned:

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

Similarly, intermediate 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 (2) and are illustrated in Figs. 8 and 9, respectively.

C. 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 (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

$$\forall t \geq 0 \quad \mathsf{dbf}(F, t) \leq \alpha(t - \Delta)_0. \tag{14}$$

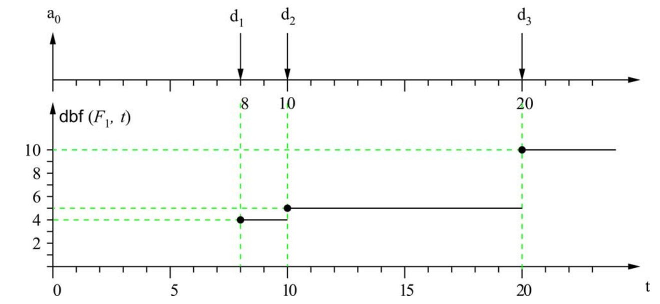

Fig. 8. Demand bound function of flow F_1.

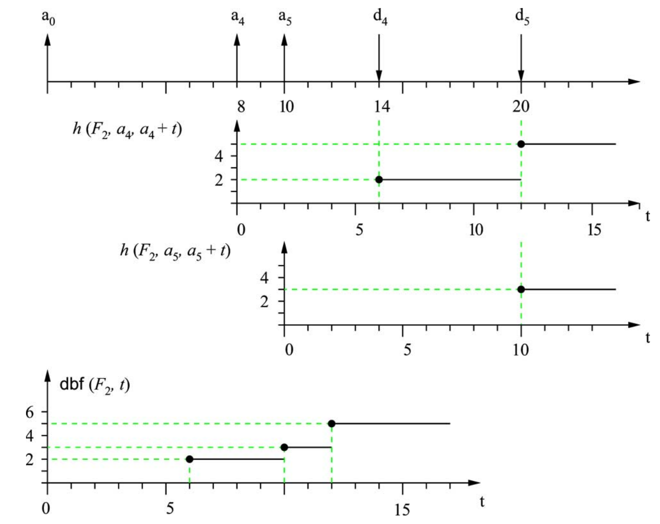

Fig. 9. Demand bound function of flow F_2.

Since the dbf is a step function, it is enough to ensure that (14) is verified at all the steps. If schedP is the set of time instants where the dbf has a step, then (14) can be equivalently ensured by

$$\forall t \in \mathsf{schedP} \quad \mathsf{dbf}(t) \leq \alpha(t-\Delta)_0. \tag{15}$$

Now, the problem is to select the (α, Δ) parameters among all possible pairs that satisfy (14). We propose to select the pair that minimizes the bandwidth B used by the virtual processor, as given by (7), which accounts for the cost of the server overhead. Hence, the best (α, Δ) pair is the solution of the following minimization problem:

$$\begin{aligned} \text{minimize} \quad & \alpha + \varepsilon\frac{1-\alpha}{\Delta} \\ \text{subject to} \quad & \mathsf{dbf}(F,t) \leq \alpha(t-\Delta)_0, \quad \forall t \geq 0, \\ & \alpha \leq 1, \end{aligned} \tag{16}$$

with $\varepsilon = 2\sigma$.

Such a minimization problem can be solved very efficiently, thanks to the good properties of both the constraint and the cost function. We first prove the convexity of the constraint.

Lemma 2: Given $t, w > 0$, let $D(t,w)$ be defined as

$$D(t,w) = \{(\alpha,\Delta) \in \mathbb{R}^2 : \alpha(t-\Delta) \geq w, \alpha \geq 0\} \tag{17}$$

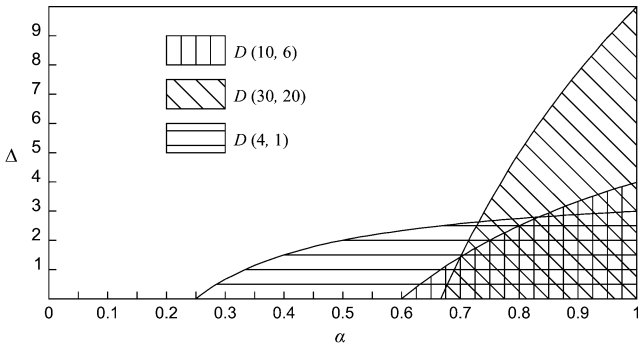

Fig. 10. Examples of the regions $D(t,w)$.

then $D(t,w)$ is convex.

Proof: We start observing that

$$\alpha(t-\Delta) \geq w \geq 0 \Rightarrow t - \Delta \geq 0 \tag{18}$$

because $\alpha \geq 0$. To prove the convexity of $D(t,w)$ we use the property that

$$\{(x,y) : f(x) \leq y\} \text{ is convex} \Leftrightarrow \frac{\mathrm{d}^2 f}{\mathrm{d}x^2} \geq 0. \tag{19}$$

In fact, we have

$$D(t,w) = \left\{ \frac{w}{t-\Delta} \leq \alpha \right\}.$$

Now

$$\frac{\mathrm{d}^2}{\mathrm{d}\Delta^2}\frac{w}{t-\Delta} = \frac{2w}{(t-\Delta)^3} \geq 0$$

because of (18). Hence, from the property of (19), the Lemma follows. ■

Fig. 10 shows examples of the domains $D(t,w)$.

Regarding the cost function, we first recall the following definition.

Definition 4 [21, Section III-D]: A function $f : \mathbb{R}^n \to \mathbb{R}$ is called *quasi-convex* if its domain and all its sublevel sets $S_v = \{x \in \mathbf{dom} f : f(x) \leq v\}$, for $v \in \mathbb{R}$, are convex.

Notice that convexity implies quasi-convexity, but the viceversa is not true [21].

Lemma 3: The cost function of problem (16) $g : [0,1] \times (0,+\infty) \to \mathbb{R}$

$$g(\alpha,\Delta) = \alpha + \varepsilon\frac{1-\alpha}{\Delta}$$

is quasi-convex.

Proof: We first notice that the domain of g, that is $G = [0,1] \times (0,+\infty)$ is convex. From the definition of quasi-convexity, we have to prove that all the level sets

$$S_v = \left\{ (\alpha,\Delta) \in G : \alpha + \varepsilon\frac{1-\alpha}{\Delta} \leq v \right\}$$

are convex (see Fig. 11 for graphical representation). Since v is interpreted as the overall bandwidth used by the reservation, we

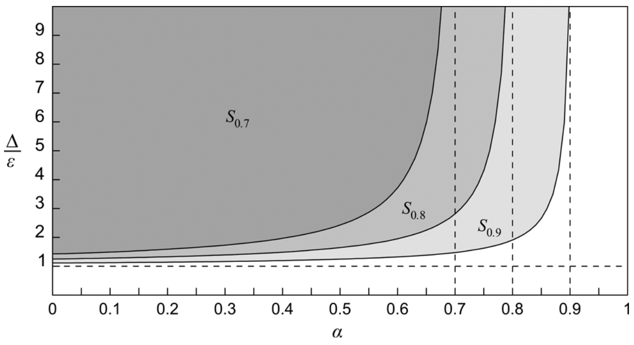

Fig. 11. Examples of the regions S_v.

only need to prove this for $v \leq 1$. If $v < \alpha$, then $S_v = \emptyset$. When $v \geq \alpha$, we have that

$$\alpha + \varepsilon \frac{1-\alpha}{\Delta} \leq v \quad \Leftrightarrow \quad \varepsilon \frac{1-\alpha}{v-\alpha} \leq \Delta$$

and from the property of (19)

$$\left\{(\alpha,\Delta) : \varepsilon \frac{1-\alpha}{v-\alpha} \leq \Delta\right\} \text{ is convex} \Leftrightarrow \frac{\mathrm{d}^2}{\mathrm{d}\alpha^2}\frac{1-\alpha}{v-\alpha} \geq 0$$

since $\varepsilon > 0$.

We have

$$\frac{\mathrm{d}}{\mathrm{d}\alpha}\frac{1-\alpha}{v-\alpha} = \frac{-1(v-\alpha)+(1-\alpha)}{(v-\alpha)^2} = \frac{1-v}{(v-\alpha)^2}$$
$$\frac{\mathrm{d}^2}{\mathrm{d}\alpha^2}\frac{1-\alpha}{v-\alpha} = 2\frac{1-v}{(v-\alpha)^3}$$

that is greater than or equal to zero, because $v \leq 1$ and $\alpha \leq v$. This proves the convexity of the level sets S_v and the quasi-convexity of g as required. ■

Since the cost function of the problem of (16) is quasi-convex (see Lemma 3) and the feasibility region is the intersection of convex regions (from Lemma 2), then the minimization problem is a standard quasi-convex optimization problem [21], which can be solved very efficiently by standard techniques. Our proposed solution is an adaptation of the dual simplex algorithm to convex problems. The details are not reported here due to lack of space. The interested reader can, however, find the details in [22].

D. 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

$$B = \sum_{k=1}^{m} B_k = \sum_{k=1}^{m}\left(\alpha_k + 2\sigma\frac{1-\alpha_k}{\Delta_k}\right). \tag{20}$$

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

$$\beta = \max_{k=1,\ldots,m} \frac{\sum_{i=k}^{m} B_i}{B_k}, \tag{21}$$

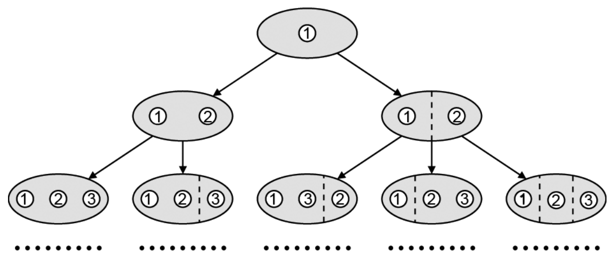

Fig. 12. The search tree.

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 [5]. In fact, in uniform multiprocessor scheduling, if $B_1 \geq B_2 \geq \cdots \geq B_m$ 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 (21), let us consider a virtual platform with m identical processors, each providing $B_k = B/m$. While the cost according to (20) is B, hence independent of the number of virtual processors, the cost according to (21) 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 Fig. 12.

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 newly 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[23], recursively computed by

$$b_{n+1} = \sum_{k=0}^{n}\binom{n}{k} b_k = \sum_{k=0}^{n}\frac{n!}{k!\,(n-k)!}\, b_k. \tag{22}$$

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

[1]Notice that in [5] the authors use $\lambda = \beta - 1$ to express the parallelism of the platform.

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

$$B_k \geq \sum_{\tau_i \in F_k} \frac{C_i}{T} > 1 \tag{23}$$

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

$$\sum_{\tau_i \in F_k} C_i > T \geq D. \tag{24}$$

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 allows pruning a subtree satisfying (23) at the highest possible level.

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

Lemma 4: In any feasible partitioning, the number of flows satisfies

$$m \geq \left\lceil \frac{C^s}{D} \right\rceil. \tag{25}$$

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

$$\frac{\sum_{\tau_i \in F_k} C_i}{D} \leq 1. \tag{26}$$

Adding (26) for all the flows, we have

$$\frac{\sum_k \sum_{\tau_i \in F_k} C_i}{D} = \frac{C^s}{D} \leq m.$$

Since m is an 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:

$$m_{\text{max}} = \left\lceil \delta \frac{C^s}{D} \right\rceil \tag{27}$$

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 $\delta \leq 2$.

```
$V_{opt} \leftarrow$ MAX; $S_{opt} \leftarrow \emptyset$;
function SEARCH_PARTITION($\Gamma$, goal)
    SEARCH($\tau_1, \emptyset, \Gamma$, goal);
end function

function SEARCH($\tau_i, S, \Gamma$, goal)
    $F_{new} \leftarrow \emptyset$;
    for each $F_k$ in $S \cup F_{new}$
        add $\tau_i$ to $F_k$;
        if $B_k \leq 1$
            if $F_k = F_{new}$
                if NUM($S \cup F_{new}$) $> m_{max}$
                    break;
                else
                    $S \leftarrow S \cup F_{new}$;
                end if
            end if
            if $i$ = NUM($\Gamma$)
                if goal = $BANDWIDTH$
                    $V \leftarrow B$ as in Eq. (20);
                else if goal = $FRAGMT$
                    $V \leftarrow \beta$ as in Eq. (21);
                end if
                if $V < V_{opt}$
                    $V_{opt} \leftarrow V$;
                    $S_{opt} \leftarrow S$;
                end if
            else
                SEARCH($\tau_{i+1}, S, \Gamma$, goal);
            end if
        end if
        remove $\tau_i$ from $F_k$;
    end for
end function
```

Fig. 13. Pseudocode of the algorithm for finding the optimal partition.

The pseudocode of the branch and bound algorithm with pruning conditions is shown in Fig. 13. In the pseudocode, S represents a partition, i.e., a set with flows F_k as its elements and $\mathbf{NUM}(S)$ gives the cardinality of the set S. S_{opt} is the optimal partition and V_{opt} is the optimal value of the objective, being either bandwidth B or fragmentation β. The objective is selected by using the argument goal having a value of either BANDWIDTH or FRAGMT. Besides, B_k, B and β are calculated using (7), (20), and (21), respectively.

IV. Heuristic Partitioning

In spite of the pruning conditions, the complexity of the branch and bound algorithm is still exponential and limits its applicability to applications consisting of no more than 15–20 tasks. To deal with larger applications, in this section we propose two polynomial-time heuristic algorithms for partitioning a real-time application into a set of virtual processors. The two methods will be later compared by simulations to evaluate their performance and runtime cost.

1) First Heuristic Algorithm: The main idea behind the first heuristic algorithm (H1) is that, if M_{low} is the lower bound on the number of cores needed by the application, then the application must contain at least M_{low} flows. Hence, the algorithm starts constructing M_{low} flows using the longest paths in the precedence graph. At the beginning, the critical path is inserted into flow F_1. Then, the algorithm tries to fit the critical path of the remaining graph into one of the existing flows. If this is not possible (i.e., the resulting schedule is not feasible), the current critical path is put into a new flow. The procedure is repeated

```
function HEURISTIC($\Gamma$, nPaths)
    $\Gamma_r \leftarrow \Gamma; S \leftarrow \emptyset$;
    if $D > C^s$
        $S \leftarrow \{\Gamma\}$;
        return $S$;
    end if
    while NUM($S$) $\le$ nPaths
        let $P$ be the critical path of $\Gamma_r$;
        FITorNEW($P$, $S$);
        $\Gamma_r \leftarrow \Gamma_r \backslash P$;
    end while
    for each $\tau_i$ in $\Gamma_r$
        FITorNEW($\{\tau_i\}$, $S$);
    end for
    return $S$;
end function

function FITorNEW($P$, $S$)
    for each $F \in S$
        $F' \leftarrow F \cup P$;
        if $F'$ is schedulable
            $S \leftarrow S \setminus F \cup \{F'\}$;
            return
        end if
    end for
    $S \leftarrow S \cup \{P\}$;
end function
```

Fig. 14. Pseudocode of the heuristic algorithms.

until M_{low} flows are created. Then, the remaining tasks in the graph are selected by decreasing computation times and put in the existing flows using a Best Fit policy. If the schedulability cannot be guaranteed within any of the existing flows, a new flow is created. Note that M_{low} can be computed as

$$M_{\text{low}} = \max\{n_h, \lceil U \rceil\},$$

where n_h is the number of *heavy tasks* (i.e., tasks having $U_i = C_i/D > 0.5$) and $U = C^s/D$.

2) Second Heuristic Algorithm: A second heuristic algorithm (H2) starts with a single flow F_1 containing the critical path and then proceeds as before by selecting the task with the longest computation time and inserting it into one of the current flows using Best Fit. If the schedulability cannot be guaranteed within any of the existing flows, a new flow is created.

3) A Naif Algorithm: For the sake of completeness, besides the two proposed heuristics (H1 and H2), a third naif heuristic algorithm is used for comparison. Such a naif algorithm builds the various flows putting the remaining tasks one by one using Next Fit, without considering precedence relations and computation times. The Naif algorithm is not part of the literature and has been introduced to evaluate the improvement of the proposed heuristics with respect to a very simple method. In this way, it is possible to position the performance of the heuristics not only with respect to optimality, but also with respect to a very simple approach.

The pseudocode of the heuristic algorithms is shown in Fig. 14, where the $\backslash$ operator denotes the set subtraction and S is the resulting partition. Notice that the only difference between the proposed heuristics is in the value of nPaths, i.e., the number of flows to be filled with critical paths before best fitting individual tasks. In particular, nPaths is set to M_{low} for H1, to 1 for H2 and to 0 for the naif algorithm.

From the pseudocode, it is clear that the complexity of the heuristic algorithms is polynomial in the number of tasks. In

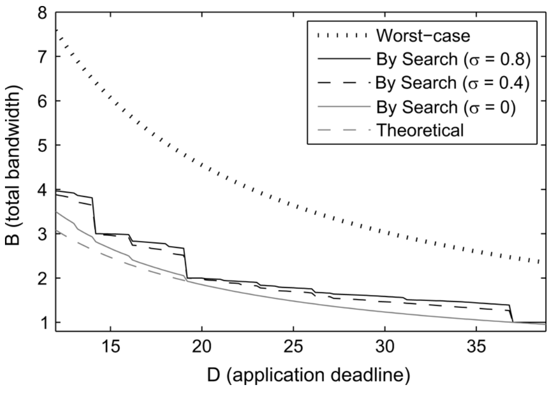

Fig. 15. Total bandwidth as a function of the application deadline.

fact, each loop executes at most n times and the feasibility check inside the FITorNEW function has $O(n^2)$ complexity, since the maximum in the dbf function [see (2)] is performed at most for all activation times and deadlines.

V. Experimental Results

To illustrate the effectiveness of the proposed search algorithm, this section presents 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 runtime). For the sake of generality, all timing parameters are expressed in generic *time units*, which can be millisecond, microsecond, or number of clock cycles.

A. Evaluation of the Partition Method

To evaluate the performance of the proposed optimal partition method, several experiments were conducted.

In a first experiment, we considered the application shown in Fig. 5, 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 4).

Fig. 15 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 (20). 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.

Considering the second optimization goal, expressed by the cost function reported by (21), Fig. 16 reports the optimal β achieved by the search algorithm, as a function of the application deadline, for different values of σ.

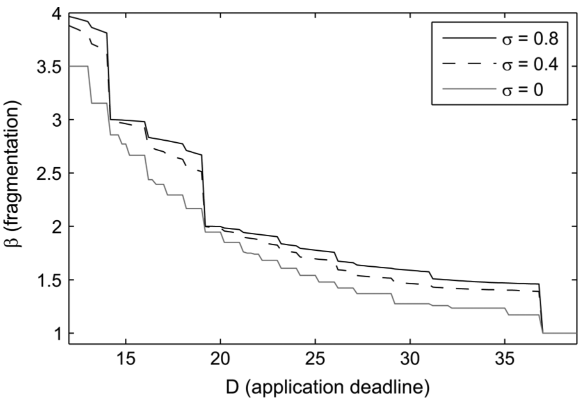

Fig. 16. β as a function of the application deadline.

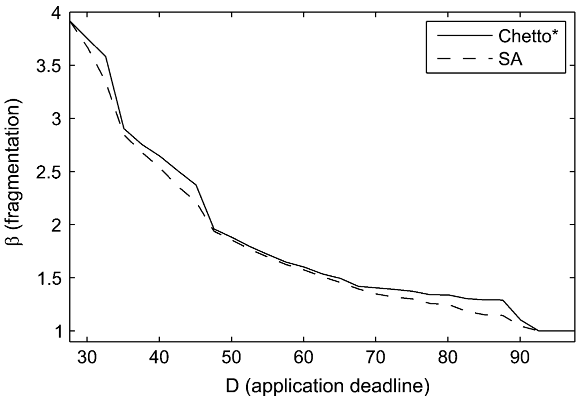

Fig. 18. Comparison of *Chetto** and *SA*.

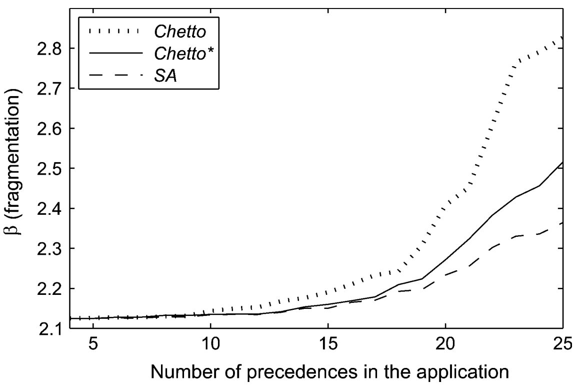

Fig. 17. β as a function of precedence number.

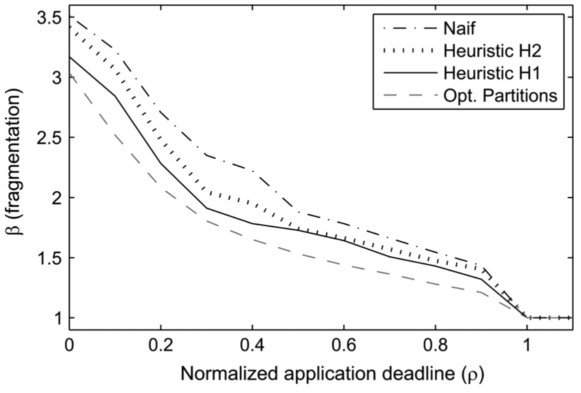

Fig. 19. Degree of fragmentation β achieved by the algorithms.

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 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 and the context switch overhead σ was set to 0.1.

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 ten simulations (differing on the sequence of random precedence link deletions). The results of this first test are reported in Fig. 17, 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.

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 Fig. 18 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.

B. Evaluation of the Heuristics

To illustrate the effectiveness of the proposed heuristics, this section presents a number of experiments aimed at comparing the performance of the heuristic algorithms.

In this experiment, we considered the application with $n = 9$ tasks illustrated in Fig. 5. The worst-case execution time C_i of each task was generated as a uniform random variable in $[1, 10]$ and the context switch overhead σ was set to 0.1 time units.

Fig. 19 reports the averaged bandwidth β (over 60 repetitions) achieved by all algorithms as a function of the normalized deadline ρ, defined as $\rho = (D - C^p)/(C^s - C^p)$. The results indicate that, in most cases, the β achieved by the first heuristic algorithm (H1) achieves better results than H2 and is close to the one of the optimal partition. In addition, both heuristics perform better than the naif approach.

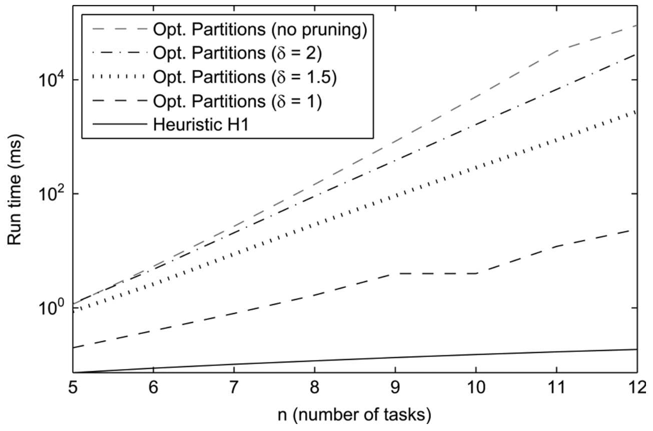

Fig. 20. Runtime of the algorithms.

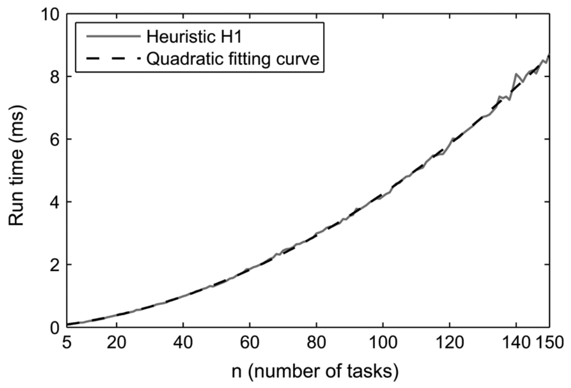

Fig. 21. Runtime of H1.

C. Evaluation of the Complexity

To test the runtime behavior of the search algorithm and the heuristics, as well as the efficiency of the pruning rule, we ran another experiment considering a fully parallel application (i.e., without precedence relations) with random computation times generated with uniform distribution in $[1, 10]$. The application deadline was set equal to $D = (C^p + C^s)/2$ and the context switch overhead was set to $\sigma = 1.6$. For each method, the runtime was measured in milliseconds as a function of the number of tasks. The branch and bound algorithm was measured for different values of the pruning parameter δ. The results of this experiment are shown in Fig. 20, 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. Furthermore, the heuristic algorithm significantly reduces the running time compared with the optimal search, even when an aggressive pruning condition $(\delta = 1)$ is used.

In a final experiment, we tested heuristic H1 for a large n. As shown in Fig. 21, the runtime grows as n^2, confirming the polynomial-time complexity of the algorithm.

VI. Related Work

The problem of partitioning a task set over a set of processing elements has been considered by many researchers under different models and assumptions.

Peng and Shin [6] proposed two branch-and-bound algorithms for partitioning a set of periodic real-time tasks, described by a task graph, on a distributed system. The method is optimal in the sense that it minimizes the maximum normalized task response time, but no bandwidth requirements were taken into account. Ramamritham [7] proposed a heuristic search to allocate and schedule periodic tasks in a distributed system, taking precedence, communication and replication requirements into account. Abdelzaher and Shin [8] presented an approach to partition large real-time applications on heterogeneous distributed systems. The scalability of the proposed method was achieved by heuristically clustering processors and tasks and assigning the clustered tasks to clustered processors in a recursive way.

The main difference with respect to the work presented here is that all the algorithms mentioned above did not consider resource requirements into account, whereas the proposed algorithm can achieve a feasible partition while minimizing either the overall application bandwidth requirements or the total number of processors.

Baruah and Fisher [24] proposed a heuristic to partition a set of deadline-constrained sporadic tasks in a multiprocessor system, but no precedence constraints were considered.

Otero *et al.* [25] 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.* [26] 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.

Bertogna *et al.* [27] considered the problem of executing a set of independently designed and validated applications upon a common platform under resource constraints. Each application is characterized by a virtual processor speed α, jitter tolerance Δ and a resource holding time H. However, no precedence constraints are considered and the underlying platform is composed of a single processor and shared resources, rather than multiple processors.

Schranzhofer *et al.* [28] presented a method for allocating tasks to a multiprocessor platform, aiming at minimizing the average power consumption. However, the application is modeled without considering timing constraints.

Leontyev and Anderson [29] proposed a multiprocessor scheduling scheme for supporting hierarchical reservations (containers) that encapsulate hard and soft sporadic real-time tasks. Recently, Bini *et al.* [9] proposed to abstract a set of m virtual processors by the set of the m supply functions [16], [17], [19] of each virtual processors. In this paper, we borrow such an abstraction of a virtual multicore platform. In all these works, however, the application is modeled as a collection of sporadic tasks and no precedence relations are taken into account.

A more accurate task model (generalized multiframe task) considering conditional execution flows, expressed by DAG, has been proposed by Baruah *et al.* [30]. 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.* [20], 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 [31] and various heuristic algorithms have been proposed in the literature to reduce the complexity [32]–[34], 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 [32], [33], is based on proper priority assignments to meet the application constraints. Another technique, called Critical Path Heuristics [31], [34], 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 interprocessor 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.* [35] 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 [36] 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 [37] 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 [38] 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, but partitioning issues and resource reservations were not addressed.

VII. Conclusion and Future Work

This paper presented a general methodology for allocating a parallel real-time application to a multicore 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.

The optimal reservation parameters (α, Δ) of each flow have been computed by solving a quasi-convex optimization problem, resulting in a minimum bandwidth on the virtual processor that hosts the flow.

Given the high complexity of the algorithm, two heuristics have also been proposed to deal with real-time applications with a large number of tasks. Simulation experiments showed that the heuristic algorithms significantly reduces the running time compared with the optimal search, even when an aggressive pruning condition (low δ) is used.

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 [37].

We also plan to integrate the communication cost into the system model. This should be possible since adding communication costs to precedences in the DAG will only affect the deadlines and activations of the tasks, while the calculation of the Demand Bound Function and the optimization of the (α, Δ) of each flow remain unchanged.

References

[1] S. Schliecker, M. Negrean, and R. Ernst, "Response time analysis on multicore ECUs with shared resources," *IEEE Trans. Ind. Informat.*, vol. 5, no. 4, pp. 402–413, Nov. 2009.

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

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

[4] L. Abeni, L. Palopoli, C. Scordino, and G. Lipari, "Resource reservations for general purpose applications," *IEEE Trans. Ind. Informat.*, vol. 5, no. 1, pp. 12–21, Feb. 2009.

[5] S. Funk, J. Goossens, and S. Baruah, "On-line scheduling on uniform multiprocessors," in *Proc. 22nd IEEE Real-Time Syst. Symp.*, London, U.K., Dec. 2001, pp. 183–192.

[6] D. T. Peng and K. G. Shin, "Static allocation of periodic tasks with precedence constraints in distributed real-time systems," in *Proc. 9th Int. Conf. Distrib. Comput. Syst.*, Newport Beach, CA, USA, Jun. 1989, pp. 190–198.

[7] K. Ramamritham, "Allocation and scheduling of precedence-related periodic tasks," *IEEE Trans. Parallel and Distrib. Syst.*, vol. 6, pp. 412–420, Apr. 1995.

[8] T. F. Abdelzaher and K. G. Shin, "Period-based load partitioning and assignment for large real-time applications," *IEEE Trans. Comput.*, vol. 49, no. 1, pp. 81–87, Jan. 2000.

[9] E. Bini, G. C. Buttazzo, and M. Bertogna, "The multi supply function abstraction for multiprocessors," in *Proc. 15th IEEE Int. Conf. Embedded and Real-Time Comput. Syst. Appl.*, Beijing, China, Aug. 2009, pp. 294–302.

[10] A. K. Mok, X. Feng, and D. Chen, "Resource partition for real-time systems," in *Proc. 7th IEEE Real-Time Technol. Appl. Symp.*, Taipei, Taiwan, May 2001, pp. 75–84.

[11] G. Buttazzo, E. Bini, and Y. Wu, "Partitioning parallel applications on multiprocessor reservations," in *Proc. 22nd Euromicro Conf. Real-Time Syst.*, Bruxelles, Belgium, Jul. 2010, pp. 24–33.

[12] D. Cancila, R. Passerone, T. Vardanega, and M. Panunzio, "Toward correctness in the specification and handling of non-functional attributes of high-integrity real-time embedded systems," *IEEE Trans. Ind. Informat.*, vol. 6, no. 2, pp. 181–194, May 2010.

[13] G. Buttazzo, "Achieving scalability in real-time systems," *IEEE Comput.*, vol. 39, no. 5, pp. 54–59, May 2006.

[14] 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 Syst.*, vol. 2, pp. 301–324, 1990.

[15] A. Rahni, E. Grolleau, and M. Richard, "Feasibility analysis of non-concrete real-time transactions with EDF assignment priority," in *Proc. 16th Conf. Real-Time and Network Syst.*, Rennes, France, Oct. 2008, pp. 109–117.
[16] G. Lipari and E. Bini, "Resource partitioning among real-time applications," in *Proc. 15th Euromicro Conf. Real-Time Syst.*, Porto, Portugal, Jul. 2003, pp. 151–158.
[17] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proc. 24th Real-Time Syst. Symp.*, Cancun, Mexico, Dec. 2003, pp. 2–13.
[18] A. Easwaran, M. Anand, and I. Lee, "Compositional analysis framework using EDP resource models," in *Proc. 28th IEEE Int. Real-Time Syst. Symp.*, Tucson, AZ, 2007, pp. 129–138.
[19] X. Feng and A. K. Mok, "A model of hierarchical real-time virtual resources," in *Proc. 23rd IEEE Real-Time Syst. Symp.*, Austin, TX, Dec. 2002, pp. 26–35.
[20] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *Real-Time Syst.*, vol. 2, no. 3, pp. 181–194, Sep. 1990.
[21] S. Boyd and L. Vandenberghe*, Convex Optimization*. Cambridge, U.K.: Cambridge Univ. Press, 2004.
[22] E. Bini, G. Buttazzo, and Y. Wu, "Selecting the minimum consumed bandwidth of an EDF task set," in *Proc. 2nd Workshop on Compositional Real-Time Syst.*, Washington, DC, Dec. 2009.
[23] G. Rota, "The number of partitions of a set," *Amer. Math. Monthly*, vol. 71, no. 5, pp. 498–504, 1964.
[24] S. Baruah and N. Fisher, "The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems," *IEEE Trans. Comput.*, vol. 55, no. 7, pp. 918–923, Jul. 2006.
[25] C. P. Otero, 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. New York: Springer, 2006, ch. 5, pp. 109–137.
[26] I. Shin, A. Easwaran, and I. Lee, "Hierarchical scheduling framework for virtual clustering multiprocessors," in *Proc. 20th Euromicro Conf. Real-Time Syst.*, Prague, Czech Republic, Jul. 2008, pp. 181–190.
[27] M. Bertogna, N. Fisher, and S. Baruah, "Resource-sharing servers for open environments," *IEEE Trans. Ind. Informat.*, vol. 5, no. 3, pp. 202–219, Aug. 2009.
[28] A. Schranzhofer, J.-J. Chen, and L. Thiele, "Dynamic power-aware mapping of applications onto heterogeneous MPSoC platforms," *IEEE Trans. Ind. Informat.*, vol. 6, no. 4, pp. 692–707, Nov. 2010.
[29] H. Leontyev and J. H. Anderson, "A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees," in *Proc. 20th Euromicro Conf. Real-Time Syst.*, Prague, Czech Republic, Jul. 2008, pp. 191–200.
[30] S. K. Baruah, D. Chen, S. Gorinsky, and A. K. Mok, "Generalized multiframe tasks," *Real-Time Syst.*, vol. 17, no. 1, pp. 5–22, Jul. 1999.
[31] V. Sarkar*, Partitioning and Scheduling Parallel Programs for Multiprocessors*. Cambridge, MA: MIT Press, 1989.
[32] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Commun. ACM*, vol. 17, no. 12, pp. 685–690, 1974.
[33] H. El-Rewini and T. G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *J. Parallel and Distrib. Comput.*, vol. 9, no. 2, pp. 138–153, 1990.
[34] Y. k. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Trans. Parallel and Distrib. Syst.*, vol. 7, no. 5, pp. 506–521, May 1996.
[35] S. Collette, L. Cucu, and J. Goossens, "Integrating job parallelism in real-time scheduling theory," *Information Process. Lett.*, vol. 106, no. 5, pp. 180–187, May 2008.
[36] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, 1987.
[37] P. Jayachandran and T. Abdelzaher, "Delay composition algebra: A reduction-based schedulability algebra for distributed real-time systems," in *Proc. 29th IEEE Real-Time Syst. Symp.*, Barcelona, Spain, Dec. 2008, pp. 259–269.
[38] N. Fisher and S. Baruah, "The feasibility of general task systems with precedence constraints on multiprocessor platforms," *Real-Time Syst.*, vol. 41, no. 1, pp. 1–26, 2009.

Giorgio Buttazzo (SM'05) graduated with a Degree in electronic engineering from the University of Pisa, Pisa, Italy, in 1985. He received the M.S. degree in computer science from the University of Pennsylvania, Philadelphia, in 1987 and the Ph.D. degree in computer engineering from the Scuola Superiore Sant'Anna of Pisa, in 1991.

He is a Full Professor of Computer Engineering at the Scuola Superiore Sant'Anna of Pisa. From 1987 to 1988, he worked on active perception and real-time control at the G.R.A.S.P. Laboratory, University of Pennsylvania. He has authored six books on real-time systems and over 200 papers in the fields of real-time systems, robotics, and neural networks.

Prof. Buttazzo has been Program Chair and General Chair of major international conferences on real-time systems. He is Editor in Chief of *Real-Time Systems*, Associate Editor of IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, and Chair of the IEEE Technical Committee on Real-Time Systems.

Enrico Bini (M'09) received the Laurea degree in computer engineering from the Università di Pisa, Pisa, Italy, in 2000 and the Ph.D. degree in computer engineering from the Scuola Superiore Sant'Anna in Pisa in October 2004, and completed graduate studies in Mathematics in January 2010.

He is an Assistant Professor at the Scuola Superiore Sant'Anna in Pisa. In 2003 he was a visiting student at University of North Carolina at Chapel Hill, collaborating with Prof. S. Baruah. He has published more than 50 papers on scheduling algorithms, real-time operating systems, sensitivity analysis in embedded systems design, and optimization techniques.

Yifan Wu (M'10) received the B.E. and M.E. degrees in control science and engineering from Zhejiang University, Zhejiang, Hangzhou, China, in 2003 and 2006, respectively, and the Ph.D. degree in computer engineering from the Scuola Superiore Sant'Anna of Pisa, Pisa, Italy, in January 2010.

He is an Assistant Professor at Hangzhou Dianzi University, Hangzhou, China. In 2009, he was a visiting student at the Department of Automatic Control, Lund University. His research interests include real-time control systems, scheduling algorithms, resource management, and dataflow programming model.