*Proc. of the First Int.l Workshop on Adaptive Resource Management (WARM 2010), Stockholm, Sweden, April 12, 2010.*

# Heuristics for Partitioning Parallel Applications on Virtual Multiprocessors

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

***Abstract*—The problem of partitioning a parallel application on a parallel machine optimizing the available resources has been proved to be NP-hard in the strong sense. In this paper, we propose a polynomial-time heuristic algorithm for allocating a real-time application consisting of a set of tasks with precedence relations on a multi-core platform. To make the proposed method independent of a specific architecture, the allocation is performed on a set of virtual processors, achieved through a set of uniprocessor reservations. The performance of the proposed heuristics is evaluated through simulation experiments against the optimal partioning (implemented through a branch and bound algorithm) and a naif approach.**

## I. INTRODUCTION

Multi-core platforms represent a promising challenge for next generation embedded systems, since they provide a means for increasing the computational capacity with a contained power dissipation. However, new programming and design methodologies are required to fully exploit the available resources. Moreover, when a complex system is composed by different software components, resource reservation techniques are highly desired for isolating the temporal behavior of each module. On a uniprocessor system, resource reservation [15], [1] is a technique able to use a fraction of a processor as a virtual processor with reduced speed.

In a multiprocessor environment, however, the meaning of reservations has to be revisited, and different resource models have been proposed in the literature to describe the computational power supplied by a parallel machine. Such abstractions are called *virtual multiprocessors*. Funk et al. [11] proposed to abstract a set of virtual processors by their speed only. However, this abstraction cannot be practically implemented, because it requires an infinitely small reservation period. Shin et al. [20] introduced the period in the abstraction of the virtual multiprocessor. Leontyev and Anderson [13] proposed to abstract a virtual multiprocessor by the cumulative speed that it provides. Bini et al. [5] proposed to abstract a virtual multiprocessor with a maximum degree of parallelism $m$ with a set of $m$ cumulative supply functions, called *parallel supply functions*.

When the application has to be partitioned into different processors, each portion of the application is required to execute on the same physical core, to avoid migration costs. For this reason, in this paper we abstract a virtual multiprocessor by the *multi-supply function* [7], which is a collection of $m$ single-processor supply functions [10], [14], [21], each linked to a physical processor.

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

Partitioning and scheduling tasks with precedence constraints onto a multiprocessor system has been shown to be NP-Complete in general [19], and various heuristic algorithms have been proposed in the literature to reduce the complexity [2], [9], [12], but their objective was to minimize the total completion time of the task set, rather than guaranteeing timing constraints under temporal isolation.

Baruah and Fisher [3] proposed a heuristic to partition a set of deadline-constrained sporadic tasks in a multiprocessor system, but no precedence constraints were considered.

In this paper, we propose a method for allocating a real-time application consisting of a set of tasks with precedence relations on a multi-core platform. To achieve modularity and portability on different architectures, the allocation is performed on top of a virtual platform, abstracted by the Multi Supply Function (MSF) [7]. Also, to be independent of a particular reservation algorithm, each virtual processor reservation is expressed by a bounded-delay time partition, denoted by the pair $(\alpha, \Delta)$, where $\alpha$ is the allocated bandwidth and $\Delta$ is the maximum service delay. This method, originally proposed by Mok et al. [16], is general enough to express several types of resource reservation servers.

To better exploit the parallelism available in the computing platform, the application 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 determining the set of flows that optimizes the available resources is NP-hard in the strong sense, we propose a polynomial-time heuristic algorithm to partition the application into flows. The performance of the proposed heuristics is evaluated through simulation experiments against a branch and bound technique and a naif approach.

The rest of the paper is organized as follows. Section II presents the system model and the notation used throughout the paper; Section III describes the proposed approach to partition the application into flows; Section IV illustrates some experimental results to evaluate the proposed algorithms; and Section V states our conclusions.

## II. System model

We consider an application $\Gamma$ consisting of 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 $T$ 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.

Each task $\tau_i$ can be preempted at any time and is characterized by a worst-case execution time $C_i$, a deadline $d_i$ and an activation time $a_i$, both relative to the activation time of the application. Tasks are scheduled by EDF.

Notation $\tau_i \prec \tau_j$ denotes that $\tau_i$ is a *predecessor* of $\tau_j$, whereas $\tau_i \rightarrow \tau_j$ denotes that $\tau_i$ is an *immediate predecessor* of $\tau_j$. The following parameters are defined for an application.

- **Path** $P$. It is any subset of tasks $P \subseteq \Gamma$ that is 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 $C : \mathcal{P}(\Gamma) \rightarrow \mathbb{R}$ that, applied to any subset $A$ of $\Gamma$, 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:
$$C^s \stackrel{\text{def}}{=} C(\Gamma).$$
- **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
$$C^p \stackrel{\text{def}}{=} \max_{P \text{ is a path}} C(P). \tag{1}$$
Notice that the application relative deadline cannot be less than $C^p$, otherwise it is missed even on an infinite number of cores.
- **Critical path** (CP). It is a path P having $C(P) = C^p$.
- **Virtual processor** $\mathsf{VP}_k$. 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 $\mathsf{VP}_k$, which is dedicated to the execution of tasks in $F_k$ only. An application $\Gamma$ is partitioned into $m$ flows.
- **Flow computation time** $C_k^F$. It is the cumulative computation time of the tasks in flow $F_k$:
$$C_k^F \stackrel{\text{def}}{=} C(F_k).$$

## III. Proposed approach

To evaluate the effectiveness of different partitioning algorithms, the following sequence of steps is used to compute the overall cost function:

1) Partition the application into a set of flows using a specific algorithm;
2) Assign intermediate deadlines to tasks;
3) Assign intermediate activation times to tasks;
4) Compute the bandwidth required by each flow;
5) Compute the overall cost function.

Notice that partitioning the application into flows does not require modifying the tasks, since the source code for inter-task communication can be properly generated by the runtime support once flows are allocated on specific cores. The following sections explain how the different steps are performed.

### A. Partitioning an application into flows

This section describes four methods with decreasing complexity for partitioning a parallel application into a set of sequential flows. The four methods will be later compared by simulations to evaluate their performance and runtime cost.

*1) Optimal partitioning:* The optimal partitioning is implemented through a branch and bound search algorithm that explores all feasible partitions, as illustrated in Figure 1.

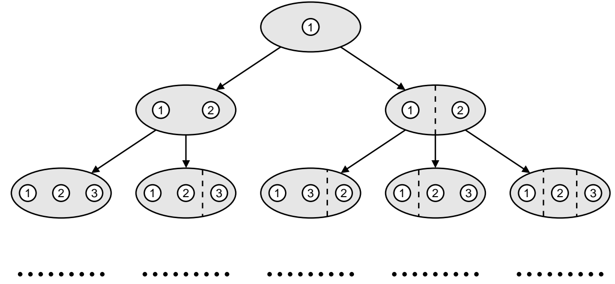

Figure 1. The search tree.

At the root level (level 1), task $\tau_1$ is associated with flow $F_1$. At level 2, $\tau_2$ 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 $\tau_i$ is assigned either to one of the existing flows, or to a new created flow. Hence, the depth of the tree is equal to $n$, the number 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 $b_n$ [18]. The average complexity of the search can be reduced by using pruning conditions to cut unfeasible and redundant branches for improving the runtime behavior of the algorithm. Whenever a node has a flow with bandwidth greater than one, the whole branch can be pruned, 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.

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

*Lemma 1:* In any feasible partitioning, the number of flows satisfies

$$m \geq \left\lceil \frac{C^s}{D} \right\rceil. \tag{2}$$

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

$$\frac{C(F_k)}{D} \leq 1. \tag{3}$$

Adding equations (3) for all the flows, we have

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

And since $m$ is integer,

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

■

We also 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:

$$m_{\max} = \left\lceil \delta \frac{C^s}{D} \right\rceil \tag{4}$$

where $\delta \geq 1$ is a parameter for tuning the size of the search tree. A value of $\delta$ close to one allows a significant improvement in terms of execution time, but at the price of losing optimality. Larger values of $\delta$ permit reaching optimality at the price of a higher execution time.

*2) First heuristic algorithm:* Although the pruning conditions, the worst-case complexity of the branch and bound algorithm is exponential and the method is practically not usable for applications larger than about 15 tasks. For this reason, here we propose a first heuristic algorithm (H1) to partition the applications into flows. The main idea behind the algorithm 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 second critical path in the remaining graph is put in a second flow $F_2$, and so on, up to flow $F_{M_{low}}$. Notice that, being $D > C^p$, all constructed flows are feasible. 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,

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

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

```
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:τ_k→τ_j} (d_j − C_j/U^p);
}
```

Figure 2. Deadline assignment algorithm.

*3) Second heuristic algorithm:* A second heuristic algorithm (H2) we propose is to starts with a single flow $F_1$ containing the critical path, and then proceeding as before selecting the task with the longest computation time and inserting it in 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.

*4) A naif algorithm:* A third naif heuristic algorithm is used just for comparison. The naif algorithm builds the various flows putting the remaining tasks one by one using Next Fit, without considering precedence relations and computation times.

### *B. Assigning intermediate deadlines*

Given a partition $\{F_1, \ldots, F_m\}$ of the application into $m$ flows, activation times $a_i$ and deadlines $d_i$ are assigned to all tasks to meet precedence relations and timing constraints. The assignment is performed according to the algorithm illustrated in Figure 2, which is a variation of the rule proposed by Chetto-Silly-Bouchentouf [8], adapted to work on multi-core systems and slightly modified to reduce the bandwidth requirements. Note that $U^p = C^p/D$.

### *C. Assigning activation times*

Activation times are set in a similar fashion, taking into account that different flows can potentially execute in parallel on different cores. Clearly, $\tau_i$ cannot be activated before all its predecessors have finished.

Let $\tau_j$ be a predecessor of $\tau_i$ and let $F_k$ be the flow $\tau_i$ belongs to. If $\tau_j \in F_k$, then the precedence constraint is already enforced by the deadline, hence it is sufficient to make sure that $\tau_i$ is not activated earlier than $\tau_j$. In general, we must ensure that

$$a_i \geq \max_{\tau_j \to \tau_i, \tau_j \in F_k} \{a_j\} \stackrel{\text{def}}{=} a_i^{\text{prec}}. \tag{5}$$

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

$$a_i \geq \max_{\tau_j \to \tau_i, \tau_j \notin F_k} \{d_j\} \stackrel{\text{def}}{=} d_i^{\text{prec}}. \tag{6}$$

In general, $a_i$ must satisfy both (6) and (5). Moreover $a_i$ should be as early as possible so that the resulting demand

bound function is minimized [4]. Hence, we set

$$a_i = \max\{a_i^{\text{prec}}, d_i^{\text{prec}}\}. \tag{7}$$

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.

### *D. Bandwidth requirements for a flow*

Once activation times and deadlines have been set for all tasks, each flow can be independently executed on a different virtual processor 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. To do that we use the concept of demand bound function [4]. The processor demand of a task $\tau_i$ in any interval $[t_1, t_2]$ is defined as the computation time $g_i(t_1, t_2)$ requested by those instances of $\tau_i$ activated in $[t_1, t_2]$ that must be completed in $[t_1, t_2]$. That is,

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

where $(x)_0$ denotes $\max\{0, x\}$. The overall demand bound function of a subset of tasks $A \subseteq \Gamma$ is

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

As suggested by Rahni et al. [17], a more compact formulation of the demand bound function can be expressed as follows:

$$\mathsf{dbf}(A, t) \stackrel{\text{def}}{=} \max_{t_1} h(A, t_1, t_1 + t). \tag{8}$$

Using Equation (8), we have that a flow $F$ is schedulable on the virtual processor $\mathsf{VP}_k$ characterized by bandwidth $\alpha_k$ and delay $\Delta_k$ if and only if:

$$\forall t \geq 0 \quad \mathsf{dbf}(F, t) \leq \alpha_k (t - \Delta_k)_0. \tag{9}$$

Now the problem is to select the $(\alpha_k, \Delta_k)$ parameters among all possible pairs that satisfy Eq. (9). We propose to select the pair that minimizes the effective bandwidth $B_k$ used by the virtual processor (including overhead). If $\sigma$ 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 $\alpha_k$ and $\Delta_k$ we have

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

Hence,

$$B_k = \alpha_k + 2\sigma \frac{1 - \alpha_k}{\Delta_k}. \tag{10}$$

Hence, the best $(\alpha, \Delta)$ pair is the solution of the following minimization problem:

$$\begin{aligned} \text{minimize} \quad & \alpha + \varepsilon \frac{1 - \alpha}{\Delta} \\ \text{subject to} \quad & \mathsf{dbf}(F, t) \leq \alpha (t - \Delta)_0, \quad \forall t \geq 0, \end{aligned} \tag{11}$$

with $\varepsilon = 2\sigma$.

This problem has been shown [6] to have a very efficient solution that exploits the convexity of the domain and the quasi-convexity of the cost function.

### *E. Optimization goals*

Two possible optimization objectives have been considered. As a first optimization goal, we considered minimizing the overall bandwidth requirement of the selected flows, that is

$$B = \sum_{k=1}^{m} B_k = \sum_{k=1}^{m} \left( \alpha_k + 2\sigma \frac{1 - \alpha_k}{\Delta_k} \right). \tag{12}$$

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

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

$$\beta = \max_{k=1,\ldots,m} \frac{\sum_{i=k}^{m} B_i}{B_k}. \tag{13}$$

The selection of this metric is inspired by the global EDF test on uniform multiprocessors [11]. 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 $\beta$ 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 (13), let us consider a virtual platform with $m$ identical processors, each providing $B_k = B/m$. While the cost according to Eq. (12) is $B$, hence independent of the number of virtual processors, the cost according to Eq. (13) is $m$. It follows that the minimization of $\beta$ leads to the reduction of number of flows in which the application is partitioned. Nonetheless, the minimization of $\beta$ 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 $\beta$ is also an upper bound of the overall bandwidth $B$, and a minimization of $\beta$ leads indirectly to the selection of a low value of $B$ as well.

[1] Note that in [11] $\lambda = \beta - 1$ is used to express the parallelism of the platform.

## IV. Experimental results

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

In a first experiment, we considered the application with $n = 9$ tasks illustrated in Figure 3. 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 $\sigma$ was set to 1.6 time units.

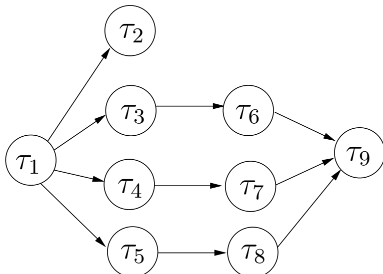


Figure 3. Sample application with 9 tasks.

Figure 4 reports the averaged bandwidth $B$ (over 60 repetitions) achieved by all algorithms as a function of the normalized deadline $\rho$, defined as $\rho = (D - C^p)/(C^s - C^p)$.

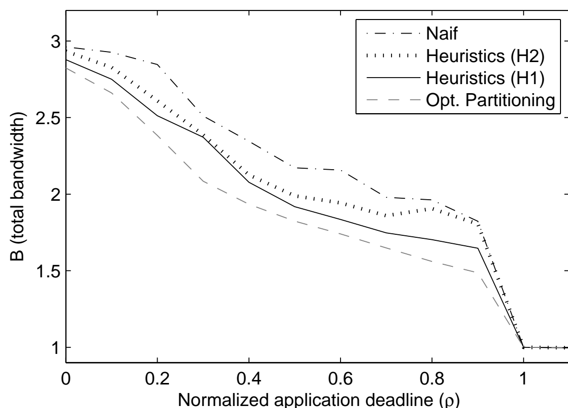


Figure 4. Bandwidth achieved by the algorithms.

The figure shows that the first heuristic algorithm (H1) achieves a bandwidth quite close to the one obtained by the optimal partitioning algorithm in most cases. Moveover, the both heuristics perform better than the naif approach.

Figure 5 reports the results of a second experiment, in which we evaluated the performance of the algorithms with respect to the second optimization goal, $\beta$, as a function of the normalized deadline $\rho$. Also in this case, each point is computed as the average over 60 repetitions. The results indicate that, in most cases, the $\beta$ achieved by the heuristic algorithms is close to the one of the optimal partition.

In a third experiment, we tested the run time of the algorithms using a fully parallel application (i.e., without

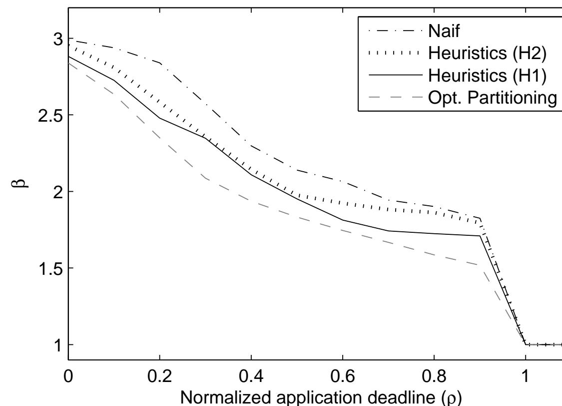


Figure 5. Degree of fragmentation $\beta$ achieved by the algorithms.

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 run time 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 $\delta$. The results of this experiment are shown in Figure 6, which clearly shows that the heuristic algorithm significantly reduces the running time compared with the optimal search, even when an aggressive pruning condition (low $\delta$) is used.

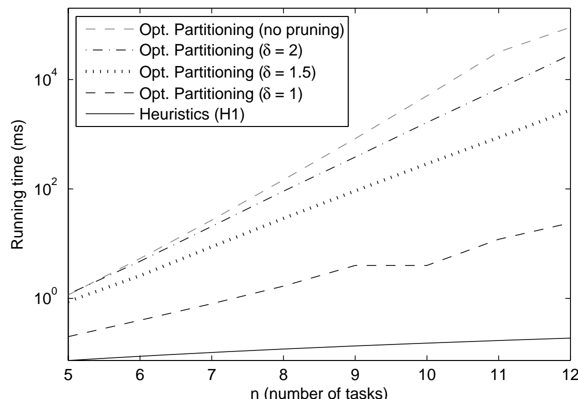


Figure 6. Run time of the algorithms.

## V. Conclusions

In this paper, we presented two heuristic algorithms 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. Independency is achieved through the concept of virtual processor, which abstracts a resource reservation mechanism

by means of two parameters, $\alpha$ (the bandwidth) and $\Delta$ (the maximum service delay).

The major contribution of this work was the development of a heuristic 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 degree of fragmentation $\beta$. The computational requirements of each flow were 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.

Simulation experiments showed the effectiveness of the proposed approach, which can be adopted to efficiently use the available resources (e.g., reducing the required bandwidth) also in large real-time applications.

## References

[1] Luca Abeni and Giorgio Buttazzo. Resource reservation in dynamic real-time systems. *Real-Time Systems*, 27(2):123–167, July 2004.

[2] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, 1974.

[3] Sanjoy Baruah and Nathan Fisher. The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems. *IEEE Transactions on Computers*, 55(7):918–923, 2006.

[4] Sanjoy K. Baruah, Rodney Howell, and Louis Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2:301–324, 1990.

[5] Enrico Bini, Marko Bertogna, and Sanjoy Baruah. Virtual multiprocessor platforms: Specification and use. In *Proceedings of the* $30^{th}$ *IEEE Real-Time Systems Symposium*, pages 437–446, Washinghton, DC, USA, December 2009.

[6] Enrico Bini, Giorgio Buttazzo, and Yifan Wu. Selecting the minimum consumed bandwidth of an EDF task set. In $2^{nd}$ *Workshop on Compositional Real-Time Systems*, Washington (DC), USA, December 2009. available at http://retis.sssup.it/~bini/publications/.

[7] Enrico Bini, Giorgio C. Buttazzo, and Marko Bertogna. The multy supply function abstraction for multiprocessors. In *Proceedings of the* $15^{th}$ *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 294–302, Beijing, China, August 2009.

[8] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2(3):181–194, September 1990.

[9] Hesham El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9(2):138–153, 1990.

[10] Xiang Feng and Aloysius K. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of the* $23^{rd}$ *IEEE Real-Time Systems Symposium*, pages 26–35, Austin, TX, U.S.A., December 2002.

[11] Shelby Funk, Joël Goossens, and Sanjoy Baruah. On-line scheduling on uniform multiprocessors. In *Proceedings of the* $22^{nd}$ *IEEE Real-Time Systems Symposium*, pages 183–192, London, United Kingdom, December 2001.

[12] Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7:506–521, May 1996.

[13] Hennadiy Leontyev and James H. Anderson. A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees. In *Proceedings of the* $20^{th}$ *Euromicro Conference on Real-Time Systems*, pages 191–200, Prague, Czech Republic, July 2008.

[14] Giuseppe Lipari and Enrico Bini. Resource partitioning among real-time applications. In *Proceedings of the* $15^{th}$ *Euromicro Conference on Real-Time Systems*, pages 151–158, Porto, Portugal, July 2003.

[15] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves for multimedia operating systems. Technical Report CMU-CS-93-157, Carnegie Mellon University, Pittsburg, May 1993.

[16] Aloysius K. Mok, Xiang Feng, and Deji Chen. Resource partition for real-time systems. In *Proceedings of the* $7^{th}$ *IEEE Real-Time Technology and Applications Symposium*, pages 75–84, Taipei, Taiwan, May 2001.

[17] Ahmed Rahni, Emmanuel Grolleau, and Michael Richard. Feasibility analysis of non-concrete real-time transactions with edf assignment priority. In *Proceedings of the* $16^{th}$ *conference on Real-Time and Network Systems*, pages 109–117, Rennes, France, October 2008.

[18] Giancarlo Rota. The number of partitions of a set. *American Mathematical Monthly*, 71(5):498–504, 1964.

[19] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, USA, 1989.

[20] Insik Shin, Arvind Easwaran, and Insup Lee. Hierarchical scheduling framework for virtual clustering multiprocessors. In *Proceedings of the* $20^{th}$ *Euromicro Conference on Real-Time Systems*, pages 181–190, Prague, Czech Republic, July 2008.

[21] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the* $24^{th}$ *Real-Time Systems Symposium*, pages 2–13, Cancun, Mexico, December 2003.