Minimizing the number of processors for real-time distributed systems

François Dorin, Michael Richard, Emmanuel Grolleau, Pascal Richard

Minimizing the number of processors for real-time distributed systems

F. DORIN, M. RICHARD, E. GROLLEAU, P. RICHARD
ENSMA - Université de Poitiers
LISI
1 rue Clément Ader, BP 40109,
86961 Chasseneuil du Poitou Cedex, France
{francois.dorin,michael.richard,emmanuel.grolleau,pascal.richard}@lisi.ensma.fr

Abstract

In this paper we present a new search method for partitioning and scheduling a set of periodic tasks on a multiprocessor or distributed architecture. The schedule is fixed-priority driven and task migration is not allowed. The aim of this algorithm is to minimize the number of processors used for scheduling a set of tasks. Moreover, we assume that the number of processors obtained by our method is optimal in respect to the holistic analysis. The paper then compares experimental results from the presented method to the FBB-FFD [9] partitioning algorithm in a multiprocessor context.

1. Introduction

Due to their potential for high performance and high reliability, distributed systems are being used for an increasing number of real-time applications. Physical architectures of such systems are composed of several processors interconnected through one or more networks like CAN[1] in automotive systems for example. The software layer is composed of tasks that communicate by exchanging messages via a communication device. Tasks are time-critical, meaning that each task must be completed by its deadline, otherwise serious consequences may ensue. This software layer has to be mapped on the hardware architecture.

In this paper we focus on the minimization of the number of processors used in multiprocessor or distributed architecture in order to require less electrical power and to reduce system cost. For some kind of systems like satellites or UAVs[2], saving energy is one of the most important point. Indeed, saving energy impacts on the weight, size and autonomy of the system. Moreover, in automotive industry, reducing the number of processors can reduce the cost of the embedded system and thus impact the cost of the vehicle.

For uniprocessor real-time systems and synchronously released fixed-priority tasks, verifying that the tasks meet their deadlines can be computed in a pseudo-polynomial time [12], but it is not known if a fully polynomial time algorithm exists. Note that no necessary and sufficient schedulability condition is known for real-time multiprocessor or distributed systems. Our method is based on the holistic analysis first presented by Tindell [29] which has been widely used [20, 27, 18, 26]. This analysis is based on the concept of busy period introduced by Lehoczky [12]. The holistic analysis is based on the knowledge of tasks allocation on the processors. In order to lead such an analysis, we need to know how to partition and how to affect priorities to the tasks set. Two algorithm families need to be considered:

- global scheduling: under global scheduling, task migration is allowed and a preempted task can be resumed on a different processor without additional cost. This assumption is only valid for multiprocessor systems, not for distributed systems where task migration cannot be assumed to have no additionnal cost. Examples of this kind of algorithm can be found in [3], [2] or [6].
- partitioned scheduling: under partitioned scheduling, tasks are allocated on processors at first and then a uniprocessor scheduling analysis is performed on each processor. In the literature, we can find several methods dealing with this kind of problem: algorithms based on bin-packing [9, 5, 14, 21], but also theoretical results as schedulability tests in the case of fixed-priority scheduling and independent tasks [4, 10]. These algorithms do not optimally minimize the number of processors but provide a sufficient condition of schedulability. For example, in [5], the au-

[1] Controller Area Network
[2] Unmanned Aerial Vehicle

thors proved that if a tasks set can be successfully partitioned on m processors, then their algorithm can partition this tasks set on m processors where each processor is $\left(4 - \frac{2}{m}\right)$ times faster. In [4], Baruah has developed an algorithm which can successfully schedule a set of tasks on m processors if the total utilization does not exceed $\frac{(m+1)}{2}$.

Nevertheless, these results cannot be applied in a dependent tasks context, that is to say for a distributed system. Allocating dependent tasks is a $\mathcal{NP}$-hard problem [13], thus there is no efficient algorithm solving the general schedulability problem for multiprocessor real-time systems. In the literature, objectives of authors works dealing with allocation and scheduling are usually either:

- To validate the application. Allocation and scheduling are usually considered as two independent stages. Most of the time, the scheduling policy is a priori known, as in [15, 25, 17, 1, 28]. These approaches mainly focus on the allocation process.
- Or to optimize one or several criteria such as the workload balancing [24], the number of required processors [16] or the response time of tasks [19, 11].

In [23], authors propose a method that simultaneously allocates tasks to processors and assigns priorities to tasks and messages. This method is based on the holistic analysis [30, 29] to verify the schedulability of tasks. In practice, there exists feasible schedules that are not validated by a holistic analysis. The method limits its search within the subset of schedules that can be validated by a holistic analysis. This method is optimal in the sense that if there exist feasible holistic schedules then our method always find one of them.

But this method needs to know exactly the hardware architecture, that is to say the number of processors in the system. The contribution of this paper relaxes this latter constraint and considers the number of processors as an output and not as an input parameter. Our algorithm is also optimal in the sense that if our algorithm completes and finds a solution using m processors, then there is no solution using less than m processors which can be validated by a holistic analysis. Our method has a very large scope; for example:

- multiprocessor systems.
- pool of multiprocessors in a distributed architecture.
- distributed system with identical processors linked through one or several network(s).

This paper is organized as follow: Section 2 introduces basic concepts and notations used in the rest of this paper. In Section 3, we explain the algorithm principles and the differences with the method presented in [23]. Section 4 provides some numerical results and a performance comparison with the FBB-FFD algorithm in a multiprocessor context.

2. Task and System Models

In this section, the characteristics and assumptions of supported hard real-time distributed systems are presented. Tasks and processors are grouped into different sets, denoted pools hereafter. All processors belonging to a pool are identical. Tasks are allocated step by step to processors of the pool in which they belong to.

2.1. Task model

A *periodic task* $\tau_i = (C_i, D_i, T_i, J_i)$ is characterized by a *worst-case execution time* C_i, a *relative deadline* D_i, a *period* T_i and a *release jitter* J_i. The required amount of computational capacity of a task τ_i is called the *task utilization* and is denoted $u_i = \frac{C_i}{T_i}$. The deadline D_i is arbitrary, that is to say it can be less, equal or greater than the task period T_i. An instance of a task is also called a *job*. A new instance is released every T_i time units. Here, we only consider fixed priority tasks. Thus, all occurrences of a tasks have the same priority, and there is no more than one task assigned to a priority level (i.e., two tasks cannot have the same priority). We also assume that preemption is allowed at no cost.

Let Tr_i denote the *worst-case response time* of the task τ_i, that is to say an upper bound of the time needed by τ_i to finish in the worst case. So, to be schedulable, the worst case response time of a task must be lower or equal to the deadline.

Let τ be a system of periodic tasks where $\tau = \{\tau_1, \ldots, \tau_n\}$ and $\tau_i = (C_i, D_i, T_i, J_i)$ for all $i, 1 \leq i \leq n$.

Let Pr_i denote the *processor* i. When a task is allocated to a processor, it has a *fixed priority* denoted π_i. When a task τ_i has a higher priority than a task τ_j, we note $\pi_i > \pi_j$. Once a task is allocated to a processor, all the occurrences of this task are executed on this processor since tasks migration is not allowed.

2.2. Message model

Distributed tasks exchange data by sending messages on networks (e.g., fieldbuses). To every message m_i is associated a worst-case transmission delay C_i and a period T_i. A deadline can be easily assigned to a message by considering deadlines of tasks that receive it. For instance, the deadline of a message is defined by the smallest quantity $D_k - C_k$,

where k is a receiver of the message m_i. Assigning deadline to every message allows to faster detect that a schedule is unfeasible without checking end-to-end deadlines.

Note that communicating tasks that are allocated to the same processor exchange data via the shared memory of the site. Thus, no message is needed for that purpose. In this case, communication is modeled by a precedence constraint between communicating tasks.

We consider that networks are composed of priority buses. Thus, messages can be modeled as tasks scheduled on a virtual non-preemptive processor: the network. Thus, the worst case response time for a message is defined like for a task, except for a blocking term which is added due to the non-preemptive policy. The blocking term corresponds to the longest message currently on the bus. The worst-case response time of a message depends on its length and of the kind of the network.

To summarize, we thus consider a preemptive schedule for tasks and a non-preemptive one for messages.

2.3 Hardware architecture

We consider distributed systems composed of sets of processors (called *pool*) and several fieldbuses.

Definition 1 *A pool of processors Pl_i is defined by:*

- *a set of m_i identical processors, denoted $Pr_k, 1 \leq k \leq m_i$. These processors are all connected to the same network. Some of them can be gateways to other networks.*
- *a set of tasks associated to the pool, denoted θ_i, to be allocated to the processors of the pool.*

Next, we present our algorithm that minimize the number of processors in every pool. To simplify the presentation, we consider that the system consists of one pool.

3. Task partitioning and scheduling

We developed a partitioning and scheduling algorithm based on the works presented in [23]. A Branch and Bound method stores feasible solutions into a search tree. Every node in the tree is a partial allocation and priority assignment of tasks. Every node corresponds to simultaneously allocating and assigning a priority to one task. Separating a node consists in exhausting all subsequent scheduling decisions. When a leaf is reached in the search tree (i.e., all scheduling decisions have been taken), then a holistic analysis allows to conclude if the corresponding solution is feasible or not. In order to limit the combinatorial explosion while enumerating scheduling decisions, evaluations

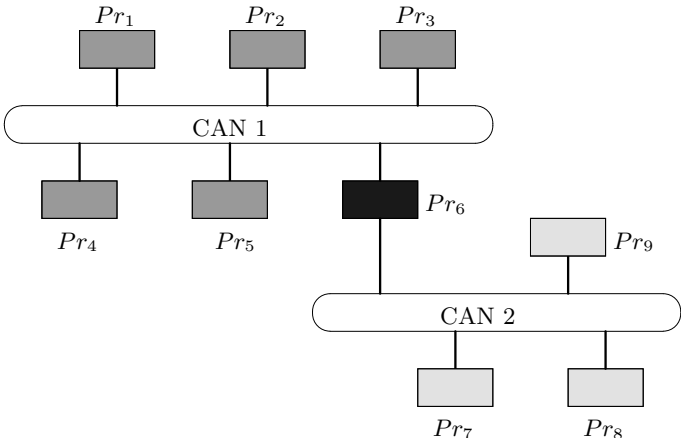

Figure 1. A real-time distributed systems with 3 pools of processors: $\{Pr1, \ldots, Pr5\}, \{Pr6\}, \{Pr7, \ldots, Pr9\}$

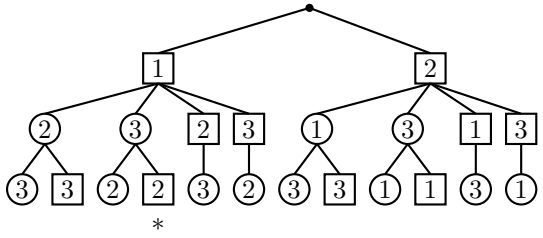

Figure 2. Example of a search tree

are performed to prune nodes that do not lead to feasible schedules.

Two kinds of vertices are defined in [8]:

- circle node: one task is allocated to the current processor to the next priority level.
- square node: one task is allocated to the highest priority on the next processor, that becomes the current processor.

Priorities are assigned to tasks from the highest priority to the lowest priority. Thus the priority of a task enumerated in a square node is the highest priority on this new processor.

Figure 2 is a search tree example. The path from the root to the leaf marked with a star represents a schedule composed of two processors. On the first processor, task τ_1 is first enumerated so it has the highest priority. Task τ_3 comes in second position and has the lowest priority on this processor. Because the node used for task τ_2 is a square node, τ_2 is allocated to a new processor which becomes the current processor.

Our algorithm tries to use all available processors in a pool and will not return a solution using less than m processors unless the number of tasks n verifies $n < m$. We extend this algorithm in order to use the minimum number of processors instead of using all the processors.

Choice of the search tree. Since the existing method uses all the available processors in the current pool, we need to extend the search tree. Considering a set of n tasks in a pool, two approaches can be considered:

1. a search tree composed of n subtrees. Each subtree enumerates all the solutions corresponding to a pool composed of i processors, $i \in [1..n]$, by using the building method presented in [23]. An example of such a search tree is shown on the first part of the Figure 3. Basically, such an approach consists in executing the algorithm presented in [23] several times with a different number of processors.

2. a new search tree structure. We can notice that in the previous search tree, a same partial solution can be evaluated several times. More precisely, the partial solution modeling the allocation of the task τ_1 at the highest priority and τ_2 at a lower priority on the same processor is represented twice in the previous search tree: first path in the first and second subtree. We propose a new search tree structure avoiding the redundancy of the same partial solutions. We can see such a search tree at the bottom of the Figure 3 for a system composed of 3 tasks allocated on 2 processors. Both search trees have the same number of leaves and represent the same set of possible schedules. The only difference is the number of nodes. In our example, the top tree contains 42 nodes and the bottom tree contains only 33 nodes. For efficiency reasons, we choose to use the bottom tree as search tree.

At each stage of the search method, the following actions are done:

1. enumeration of all the possible solutions: during this stage a task is allocated to a processor and assigned to a priority level.

2. the current solution is evaluated from a schedulability point of view and from a performance criterion point of view (i.e., minimizing the number of processors used by the current solution).

These actions are described in the next section.

In [22], it is proved that there are $\frac{n!}{m!}C_{n-1}^{m-1}$ leaves for a tree which enumerate all solutions without redoudancies for a system composed of n tasks and m processors. So, for our new search tree structure, the total number of leaves is given by equation 1:

$$\sum_{m=m_{min}}^{m_{max}} \frac{n!}{m!} C_{n-1}^{m-1} \quad (1)$$

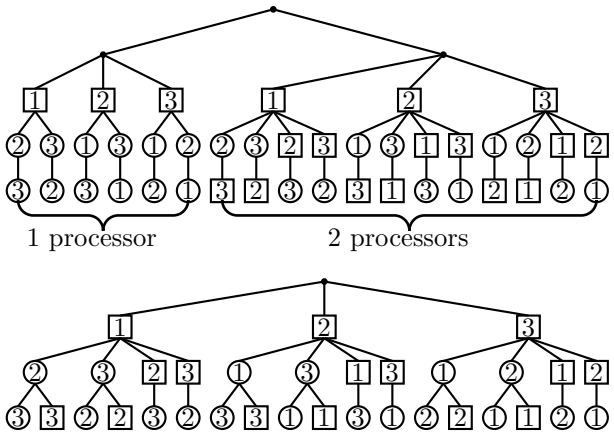

Figure 3. Two possible search trees

3.1. Enumerating solutions (branching)

During the search, at each level of the tree we have to choose which task is the next task to insert and if the task must be allocated on the current processor or on a new processor (i.e., choosing which kind of node in the search tree). This section details a set of rules used to enumerate solutions. More precisely, these rules can be categorized according to the following categories:

- building rules: necessary to avoid solution redundancies.
- cutting rules: necessary to respect the task system constraints like precedence constraints.
- branching rules: used to speed-up the search.

Building rules. The search tree corresponding to a given pool composed of n tasks is defined by the following rules avoiding redundancies in building solutions.

1. the level 0 contains a unique node which is the root of the tree.

2. the level 1 is composed of $n - m_{min} + 1$ square nodes.

3. a path, which begins at level 0 and finishes at level $i, 1 \leq i < n$, can be extended at the level $i + 1$ by any circle node or any square node among n if the rules 4,5 and 6 are respected.

4. a task τ_k can only appear once in a path.

5. a square node k cannot be used to extend a path which already contains a square node l such as $l > k$. This ensures that there will be no redundancy.

6. No path can be extended by a square node if it already contains m_{max} square nodes.

Cutting rules. Let $\prec$ denote a precedence constraint, that is to says $\tau_k \prec \tau_l$ means that the execution of τ_l cannot begin until τ_k is finished. With the following rules we check if task system constraints are satisfied. If not, the corresponding solution is not enumerated (i.e., the corresponding path is pruned and a backtrack is performed).

1. if $\tau_k \prec \tau_l$ and if τ_k and τ_l are allocated on the same processor, then it does not exist a path which begins by a square or circle node l and finishes by a circle node k and contains only circle nodes. This rule ensures that the precedence constraint is enforced.

2. A sub path beginning by a square node k and composed of l circle nodes cannot be extended by a circle node r if the load of the tasks composing the sub path, denoted by U_s, plus the load of τ_r is such as $U_s + u_r > 1$. If $U_s + u_r > 1$ and $r > k$, then a square node r is built. This rule avoids paths having one or more processors with an utilization factor higher than 1 to be built.

3. Let τ_k be the task to insert. Let the number of unallocated tasks τ_r such that $r \geq k$ be denoted by nb_k. Let nb_{Pr} denote the number of unused processors. If $nb_k > nb_{Pr}$ then the square node is not built.

4. Let U denote the load of unallocated tasks. Let nb_{Pr} denote the number of unused processors. If $U > nb_{Pr}$ then the square node is not built.

5. Let τ_k be the task to insert. If the current processor is the last and if it exists at least one predecessor of τ_k which is not already allocated, then the circle node k is not built. This insures that on the last processor the priority are coherent with the precedence constraints.

Branching strategies. Several branching rules have been implemented

- First of all, we sort tasks by non-decreasing load. Let S denote the set of unallocated tasks. At the level i, the set S contains $n-i$ items. When we allocate a task τ_k and go to the next level, the set $S^{'}$ used for the next level is $S^{'} = S - \{\tau_k\}$.

- We tried to allocate the maximum number of tasks on the current processor first. After we tried to allocate all the tasks on the current processor, we tried to allocate them on a new processor. Thus, this method tries to maximize the load of each processor before using a new one.

- a depth-first search strategy is performed in order to avoid a combinatorial explosion in space. Such a strategy ensures that the required memory to run the method is polynomially bounded in the size of the problem (see [23] for details). To speed up the method, we also perform depth-first searches in several paths of the search tree. Paths are explored one by one according to a wrapping around policy. Notice that the only data shared by the different search process is the number of processors used in the last found valid solution, allowing to implement dynamic rules. Furthermore, such an approach can be more beneficial if a parallel computer is used to run the method.

3.2. Evaluation and pruning rules (bounding)

For each inserted node, we need to check the validity of the current partial solution. More precisely, we verify the schedulability, that is to say that all tasks meet their deadline.

Valid schedule. For every leaf of the search tree, a holistic analysis [29] is led. The holistic analysis computes the worst-case response time of tasks and messages taking into account dependences between tasks and messages through a release jitter. The worst-case response times of tasks and messages can be computed by solving the following system of recurrent equations :

$$\left\{ \begin{array}{rcl} Tr_i^{(0)} & = & C_i \\ J_i^{(0)} & = & 0 \\ Tr_i^{(k)} & = & WCRT\left(\tau_i, k-1\right) \\ J_i^{(k)} & = & \min\limits_{k \in Pred(\tau_i)} \left(Tr_k\right) \end{array} \right. \quad (2)$$

where WCRT is a function which evaluates the worst-case response-time of a task τ_i and $Pred(\tau_i)$ is a function returning the set of predecessors of τ_i. The fixed-point is reached when :

$$Tr_i^k = Tr_i^{k-1} \quad (3)$$

For allocated tasks, we use the previous computation in order to evaluate their final worst case response time. Note that in the case of multiprocessor architectures and independent tasks, this computation is only done for the task allocated at the current stage. Indeed, since tasks are independent, the jitter factor is always null and a lower priority

task cannot increase the worst case response time of higher priority tasks.

For non-allocated tasks, we use the same principle to compute lower bounds (LB) of worst-case response times (Tr_i), and thus to evaluate lower bounds of release jitters. Let $G(k) = (V, E)$ be the communication graph of the current node in the search tree, then we solve the following system of recurrent equations:

$$\forall i \in V \quad \begin{cases} LB\left(Tr_i^{(k)}\right) & = & Eval\left(LB\left(J_i^{(k-1)}\right)\right) \\ LB\left(J_i^{(k)}\right) & = & Propag\left(LB\left(Tr_j^{(k)}\right)\right) \end{cases}$$

The fixed-point is the smallest positive integer p such that:

$$LB(Tr_i) = LB\left(Tr_i^{(p-1)}\right) = LB\left(Tr_i^{(p)}\right), \quad \forall i \in V$$

The $Eval$ function computes worst-case response times of tasks and messages assuming that release jitters are fixed. Then, the function $Propag$ updates release jitters according to the results obtained by $Eval$ functions. Lastly, if all tasks and messages have been allocated and assigned a priority then our evaluation process is exactly an holistic analysis.

If the evaluation process leads to a non-schedulable solution (i.e., $\exists i \in 1..n | Tr_i > D_i,\ or\ LB(Tr_i) > Di$), evaluation process is stopped and the current vertex in the search tree is pruned. Then, a backtrack is done.

Performance criterion evaluation. This set of rules is specific to our method allowing to minimize the number of processor used. Indeed, with performance rules we ensure that no solution using a number of processors higher than the number of processors used by the last found valid complete solution will be enumerated.

1. When a solution is found, then it is stored (it replaces the previous one if there is one before), and the maximum number of processors is updated with the number of processors used by the last found solution minus one.

2. When the first level is explored, let i denote the current task. The maximum number of processors is:

   - unchanged if $i \leq n - m_{max} + 1$.
   - equal to $\min(m_{max}, m_{max0} - i + 1)$.

   where m_{max0} is the maximum number of processor at the beginning of the algorithm.

To be used, performance rules must refer at an upper bound of the used processors. Moreover, our search process must know when no better solution can be found anymore. So, during the search process, the upper bound and the lower bound of the number of processors are stored, respectively denoted m_{max} and m_{min}. The bounds values evolve during the search process and have the following meaning:

- The lower bound represents the minimum number of processors require to schedule the task system, that is to say, there is no valid schedule using less than m_{min} processors. Typically, this value is initialized with the following value:

  $$m_{min} = \left\lceil \sum_{i \in \tau} u_i \right\rceil \tag{4}$$

  When the search process find that no valid schedule exists using k processors (all possible solution using k processors were enumerated with no success), this bound is updated with the new value: $m_{min} = k + 1$.

- The upper bound corresponds to the number of processors used in the last valid solution found, that is to say for which we know the system is schedulable. In others words, we can affirm there are solutions using a number of processors higher than m_{max}. At the beginning, this bound is set to the number n of tasks of the system to schedule. If a schedulable solution using k processor is found during the search, the upper bound is updated with the new value: $m_{max} = k - 1$.

At each lower or upper bound update two situations can occur:

1. $m_{max} \geq m_{min}$: the search process go on with the new updated values of bounds.

2. $m_{max} < m_{min}$: the search process completes. Indeed, no other valid schedule using m_{min} processors or less can be found. So we obtain a solution using m_{min} processors, and this one is optimal in respect to the holistic analysis.

4. Experimentations

In this section, we present experimental results and a comparison performance with the FBB-FDD algorithm [9]. Thus, we limit ourselves to independent tasks and multiprocessor architectures, since FBB-FFD is dedicated to such real-time systems.

Conditions of experimentations. Our experiments are based on the following configurations: we generate independent tasks sets using UUniFast algorithm [7] for a total load of 15.

Each configuration is composed of 50 tests and differs from the others only by the number of tasks. To be more accurate, we experiment configurations with 20, 30, 40, 60 and 100 tasks.

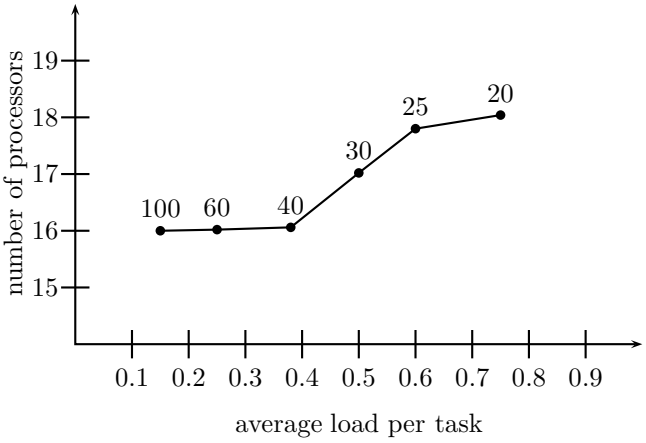

Figure 4. Number of processors in function of average load per task

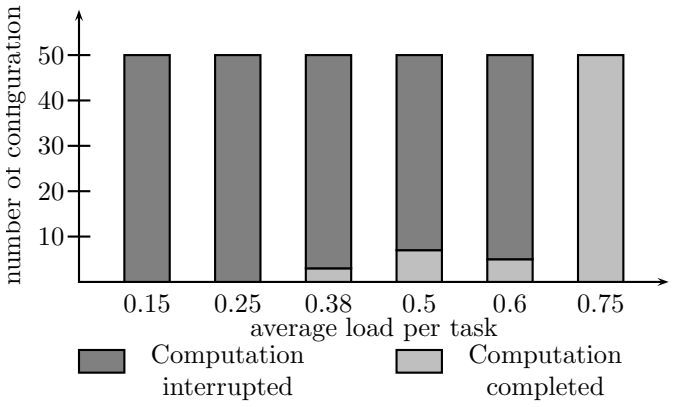

Figure 5. Number of complete computation in function of average load per task

Due to the exponential time complexity of our method, the search process requires large amount of time to finish. We stop the search after 5 minutes.

Results. Figure 4 shows the evolution of the number of processors needed to schedule a task system in function of the average load per task. Numbers close to each point represent the number of tasks of the configuration.

When average load per task is low, our algorithm finds a solution using 16 processors, for a task system which requires at least 15 processors since the total utilization of this task system is 15.

When average load per task increases, the number of processors is increasing too, since the system is more constrained when the utilization factor is high. Indeed, with a tasks set with a high utilization factor per task we have less solutions to evaluate. But, the percentage of non-schedulable solution is higher compared to a tasks set with average utilization factor per task.

Figure 5 shows the number of completed computation for each configuration in function of tasks average load. For a low average load per task, no computation has been completed before the time limit (5 minutes). Conversely, for all configurations with a high average load per task, all the computations complete. In the same way as previously, we explain this result by the fact that there are many more solutions to enumerate and evaluate in a case of low average load per task than in a case of high average load per task. A high constrained system allows us to prune current solution and backtrack very early in the search process because we are able to determine early if the current solution will lead to a better schedulable solution or not.

Due to the exponential complexity, our algorithm can take a long time to complete its execution. Two cases must be studied :

1. Since we stop the search process after 5 minutes of computation, we define a criterion to evaluate the accuracy of the solution when the computation is interrupted. Let denote an upper bound of the distance to the optimal by Δ. In theory, Δ is the difference between the number of used processors by the least schedulable solution provided by our algorithm before we stop it and the number of processors used by the optimal solution. But, in practice, the number of processors used by the optimal solution is unknown since no optimal solution is known in this context. So, we define the optimal number of processors as the number for which we know there is no solution using less processors (i.e., a lower bound). More precisely, considering a set of n tasks in a pool with a global workload U, the optimal number of processors defined previously will be:
$$N_{opt} = \lceil U/n \rceil \quad (5)$$
As a consequence, Δ is an upper bound of the distance to the optimal.

2. the search process complete: in this case, we obtain an optimal solution in respect to holistic analysis. More precisely, we can affirm that no schedulable solutions using less processor can exist. The Δ value will be null.

As shown in Figure 6 the Δ increases with the average load per task. We can explain this by the fact that, as showed

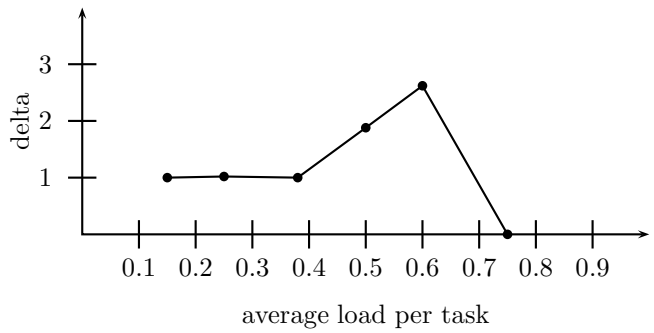

Figure 6. Evolution of the distance to the optimal

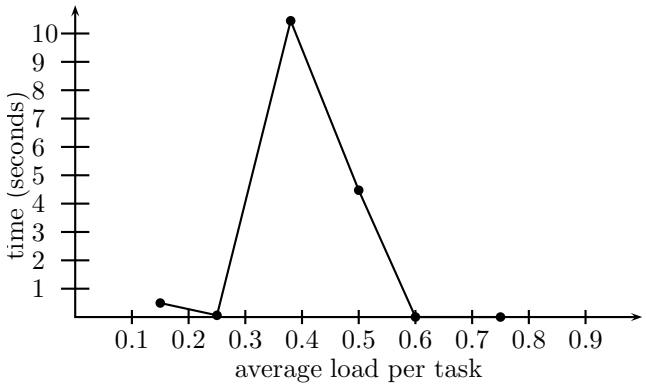

Figure 7. Evolution of computation time

on Figure 4, that the number of processors increases with the average load. We obtain a null Δ value for high average load because all the tests are completed.

Finally, Figure 7 presents the required time to find a solution. Once again we must distinguish two different cases:

1. the search process is stopped: the time considered in this case is the required computation time to find the last schedulable solution.
2. the search process terminates: the time considered here is the elapsed computation time to find the last schedulable solution (like in the previous case) plus the elapsed time to enumerate every solution using one processor less allowing us to affirm that we have obtained an optimal solution.

As we can see, our algorithm provides a solution in a very short time, especially if we take into account that the resolved problem is $\mathcal{NP}$-Hard in the strong sense.

All these experimental results show us that we can find a valid allocation and schedule using a number of processors nearly to the optimal number in a very short time. For low average load per task, a solution using 16 processors is quickly found. Moreover, when our algorithm completes, most of the computation time needed for that is spent to prove the optimality of the current solution, that is to say to evaluate all the remaining solutions. Notice that for middle average load per task, computation time increases. In this context, the search tree contains more possible schedule to explore and the computation time devoted to the schedulability test is greater too.

Comparison with FBB-FFD. We present here the results of the performance comparison between our method and the FBB-FFD algorithm. The FFB-FFD algorithm has been developed by Fisher, Baruah and Baker in [9]. This partitioning algorithm is a variant of a bin-packing heuristic known as first-fit-decreasing. Below, we briefly present this method.

Let τ denote a task system and Π a pool composed of m unit-capacity processors $\pi_1, \ldots, \pi_m$. Let assume that tasks are ordered by decreasing deadlines and assume that tasks $\tau_1, \ldots, \tau_{i-1}$ have already been allocated among the m processors. Let τ_i be the next task to be assigned. τ_i will be assigned to the first processor π_k that satisfies some conditions (cf. [9]). If there is no π_k which can satisfy conditions, then the algorithm fails and is unable to conclude that the task system is schedulable. Otherwise, it returns a valid partition(e.g., ensuring that tasks will meet their deadlines on each processor).

We monitored the following indicators: how many processors are needed by the algorithm to return a valid schedule. Instead of returning a failure when a task cannot be allocated to a processor, we add one processor. When all the tasks are allocated, we return the number of used processors.

Results of the performance comparison between our algorithm and FBB-FFD are shown in Figure 8. As we can see, our algorithm provides better solutions than FBB-FFD. Benefit is low for high average load but increases for lower average load. This figure shows that our algorithm provides a good improvement of the solution even if the time limit is reached.

5. Conclusion

In this paper, we provide an algorithm to finding an optimal solution in respect to the holistic analysis. As we have seen, the obtained solution is optimal only when our algorithm completes. Introducing a time limit allows to use our Branch and Bound algorithm as an heuristic. It produces a valid schedule using a number of processors very close to the optimal number.

The solution found is a valid schedule in the sense that

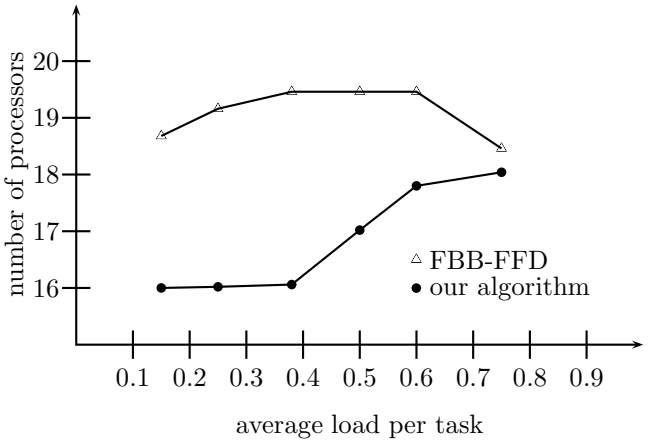

Figure 8. Comparaison with FBB-FFD

it respects all the constraints: deadlines, precedences, communications, shared resources.

Moreover, our algorithm needs a short amount of time to provide a good solution.

We can also hope that adding some constraints (like communications, precedence constraints, shared resources) will reduce the time needed by our algorithm to complete. Indeed, these new constraints can help us to detect early a non-schedulable solution.

In the future, we will try to minimize the necessary modification to an existing schedule when a transformation is applied (add a task, change the execution time or the period of a task, etc...).

References

[1] P. Altenbernd and H. Hansson. The slack method: A new method for static allocation of hard real-time tasks. *Journal of Real-Time Systems*, 13(2):103–130, Septembre 97.

[2] T. Baker. An analysis of deadline-monotonic schedulability on a multiprocessor.

[3] T. Baker. An analysis of edf schedulability on a multiprocessor. *Parallel and Distributed Systems, IEEE Transactions on*, 16(8):760–768, Aug. 2005.

[4] S. Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *Computers, IEEE Transactions on*, 53(6):781–784, June 2004.

[5] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of sporadic task systems. *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 9 pp.–, 5-8 Dec. 2005.

[6] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of edf on multiprocessor platforms. *Real-Time Systems, 2005. (ECRTS 2005). Proceedings. 17th Euromicro Conference on*, pages 209–218, 6-8 July 2005.

[7] E. Bini and G. C. Buttazzo. Biasing effects in schedulability measures. *ecrts*, 00:196–203, 2004.

[8] P. Bratley, M. Florian, and P. Robillard. Scheduling with earliest start and due date constraints on multiple machines. *Naval Research Logistic Quaterly*, 22(1):165–173, 1975.

[9] N. Fisher, S. Baruah, and T. Baker. The partitioned scheduling of sporadic tasks according to static-priorities. *Real-Time Systems, 2006. 18th Euromicro Conference on*, pages 10 pp.–, 5-7 July 2006.

[10] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. Technical Report TR01-024, 14 2001.

[11] J. Jonsson and J. Vasell. Evaluation and comparison of task allocation and scheduling methods for distributed real-time systems. In *Proceedings of the IEEE Workshop on Real-Time Applications*, pages 226–229. Montreal, Canada, 21-25 October 1996.

[12] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 201–209, 5-7 Dec 1990.

[13] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 4:237–250, 1982.

[14] J. Liebeherr, A. Burchard, Y. Oh, and S. H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Transactions on Computers*, 44(12):1429–1442, 1995.

[15] M. W. Mutka and J.-P. Li. A tool for allocating periodic real-time tasks to a set of processors. *Journal Systems Software*, 29:135–148, 1995.

[16] Y. OH and S. H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-Time System Journal*, 9(3):207–239, Novembre 1995.

[17] J. Orozco, R. Cayssials, J. Santos, and E. Ferro. Precedence constraints in hard real-time distributed systems. In *Proceedings of the 3^{rd} International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 33–38, 1997.

[18] R. Pellizzoni and G. Lipari. Holistic analysis of asynchronous real-time transactions with earliest deadline scheduling. *J. Comput. Syst. Sci.*, 73(2):186–206, 2007.

[19] D.-T. Peng, K. G. Shin, and T. F. Abdelzaher. Assignment and scheduling communicating periodic tasks in distributed real-time systems. *IEEE Transactions on Software Engineering*, 23(12), 1997.

[20] T. Pop, P. Eles, and Z. Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems, 2002.

[21] K. Ramamritham, J. Stankovic, and P.-F. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *Parallel and Distributed Systems, IEEE Transactions on*, 1(2):184–194, Apr 1990.

[22] M. Richard. *Contribution à la validation des systèmes temps réel distribués : ordonnancement à priorités fixes & placement*. PhD thesis, Université de Poitiers, 2002.

[23] M. Richard, P. Richard, and F. Cottet. Allocating and scheduling tasks in multiple fieldbus real-time systems. *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, 1:137–144, September 2003.

[24] S. Saez, J. Vila, and A. Crespo. Using exact feasibility tests for allocating real-time tasks in multiprocessor systems. In *Proceedings of the 10^{th} Euromicro Workshop on Real Time Systems*. Berlin, Germany, 17-19 June 1998.

[25] J. Santos, E. Ferro, J. Orozco, and R. Cayssials. A heuristic approach to the multitask-multiprocessor assignment problem using the empty-slots method and rate monotonic scheduling. *Journal of Real-Time Systems*, 13(2):167–199, 1997.

[26] M. Sjodin and H. Hansson. Improved response-time analysis calculations. *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 399–408, 2-4 Dec 1998.

[27] M. Spuri. Holistic analysis for deadline scheduled real-time distributed systems. Technical report, INRIA, 1996.

[28] K. Tindell, A. Burns, and A. Wellings. Allocating hard real-time tasks (an np-hard problem made easy). *Real-Time Systems*, 4(2):145–165, 1992.

[29] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocess. Microprogram.*, 40(2-3):117–134, 1994.

[30] K. W. Tindell. *Fixed Priority Scheduling of Hard Real-Time Systems*. PhD thesis, Univeristy of York, 1994.