Allocating and Scheduling Tasks in Multiple Fieldbus Real-Time Systems

Michael Richard Pascal Richard Francis Cottet
Laboratory of Applied Computer Science
LISI - ENSMA
1, avenue Clément Adler Téléport 2
BP 40109
86961 Futuroscope Cedex
France
Email: {michael.richard,pascal.richard,francis.cottet}@ensma.fr

***Abstract*— We consider real-time systems connected via several fieldbuses. Validating such systems consists in prooving that tasks meet their end-to-end deadlines. Tasks are scheduled on processors by fixed-priority schedulers. We propose an automatic method for allocating tasks on processors and assigning priorities to tasks so that every deadline is met. Allocation and scheduling are simultaneously achieved. We do not limit the search space to a specific priority rule (such as Rate Monotonic or Deadline Monotonic). Feasible schedules are validated by a Holistic Analysis. Numerical results of the method are lastly presented on a real-size application. Our tool will be a beneficial help to design real-time distributed systems.**

I. Introduction

Due to their potential for high performance and high reliability, distributed systems are being used for an increasing number of real-time applications. These applications are composed of tasks that communicate by exchanging messages via a communication device. No common memory is assumed to be available. Tasks are time-critical, meaning that each task must be completed by its deadline, otherwise serious consequences may ensue. Under this framework, fieldbuses have been developed with the specific requirements of tight real-time capabilities [1]. Fieldbuses have to strive to respect deterministic response times. For example, in automotive applications, a wildly used network is the CAN (Controller Area Network).

In [2], we proposed a method that automatically assigns priorities to tasks and messages. In the present paper, we extend this method to deals with task allocation to sets of identical processors. In many applications processors are identical and Input/Output devices can be easily connected from one to another. Some real-time functions can be allocated to several processors without any problem. Designers have to allocate these functions to sets of processors and then have to connect correctly controlled devices to processors.

Allocating tasks is a $\mathcal{NP}$-hard problem [3], thus there is no efficient algorithm to solve the schedulability problem for multiprocessor real-time systems. For uniprocessor real-time systems and synchronously released fixed-priority tasks, verifying that tasks meet their deadlines can be computed in pseudo-polynomial time [4], but it is not known if a fully polynomial time algorithm exists. In the literature, works dealing with allocation and scheduling of tasks differ from their objectives:

- To validate the application. Allocation and scheduling are usually considered as two independent stages. In many works, the scheduling policy is a priori known, as in [5], [6], [7], [8], [9]. These approaches mainly focus on the allocation process.
- To optimize the workload balancing [10], the number of used processors [11] or the response time of tasks [12], [13].

In this work, we propose a method that simultaneously allocates tasks to processors and assigns priorities to tasks and messages. Note that no necessary and sufficient schedulability condition is known for real-time distributed systems. Our method is based on the holistic analysis [14], [15] to verify that tasks are schedulable. 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. We call this subset: *holistic schedules*. Our method is optimal in the sense that if there exist feasible holistic schedules then our method always find one of them.

The next section presents software and hardware architectures supported by our method. Section 3 presents the Branch and Bound Method. Section 4 deals with numerical experimentations and lasly we conclude.

II. Real-Time Distributed Systems

We present in this section the characteristics and assumptions of supported hard real-time distributed systems. Tasks and processors are grouped into pools. All processors belonging to a pool are identical. Tasks are allocated step by step to processors of the pool in which they belong to.

A. Software architecture

Tasks can be allocated to any processor belonging to the same pool. Every task τ_i has a worst-case exectution time C_i, a deadline D_i and a period T_i between two successive releases. An occurrence of a task is called an *instance*. A task

τ_i is schedulable if its worst-case response time Tr_i is less than or equal to D_i. A schedule is feasible if, and only if, every task is schedulable.

We consider that each processor runs a real-time kernel that implements a fixed-priority scheduler. The priority of a task τ_i is denoted π_i; 0 is the highest priority level. At any time, the available task assigned to highest priority level is scheduled. The start of a task can be postponed due to input communications. This delay after the release of an instance of a task τ_i is called the release jitter and is denoted J_i. Initially, tasks are assigned to pools but they are not allocated to processors within pools and they also have not been assigned priorities.

Distributed tasks exchange data by sending messages on networks (e.g. fieldbuses). To every message m_i is associated a worst-case tranmission 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 detect faster that a schedule is unfeasible without checking end-to-end deadlines.

Communicating tasks that are allocated to the same processor exchange data via the local memory of the site. Thus, no message is needed for that purpose. Precedence relations between tasks and messages are modeled by the *communication graph*. Since our method deals with one task at each stage, then the communication graph is updated when two communicating tasks are allocated to the same processor. Let $P(i)$ be the allocated processor to the task τ_i; if τ_i is not yet allocated, then we note $P(i) = \phi$. We now formally define the communication graph: $G = (S \cup M, E)$, where $S \cup M$ is the set of vertices (S is the set of tasks, M is the set of messages) and E is the set of precedence relations between tasks and messages. Let ${}^\bullet i$ (resp. $i^\bullet$) the set of immediat predecessors (resp. successor) of a vertex i. G is a bipartite graph (an edge cannot connect two tasks or two messages together). If a task sends a message to another task that is allocated to the same processor, then the vertex corresponding to the message is deleted as well as incident edges.

Definition 1: A communication graph $G = (S \cup M, E)$, for a given allocation, is a bipartite graph that verifies the following properties :

$$S \cap M = \phi$$
$$E \subseteq (S \times M) \cup (M \times S)$$
$$\forall i \in M, \quad |{}^\bullet i| = 1$$
$$\forall i \in M, \forall k \in \{i^\bullet\} \quad \text{if} \quad P({}^\bullet i) \neq \phi \quad \text{and} \quad P(k) \neq \phi$$
$$\text{then} \quad P({}^\bullet i) \neq P(k)$$

When tasks are allocated, then vertices and edges of the communication graph can be deleted, but no new edge or vertex can be inserted while searching a feasible schedule. Hereafter we assume that tasks or messages belonging to the same connected component in the communication graph have identical periods. Such an assumption is not restrictive since if communicating tasks do not work at the same rate, then sooner or later, the slowest one will miss its deadline or an overflow will occur in a buffer of messages.

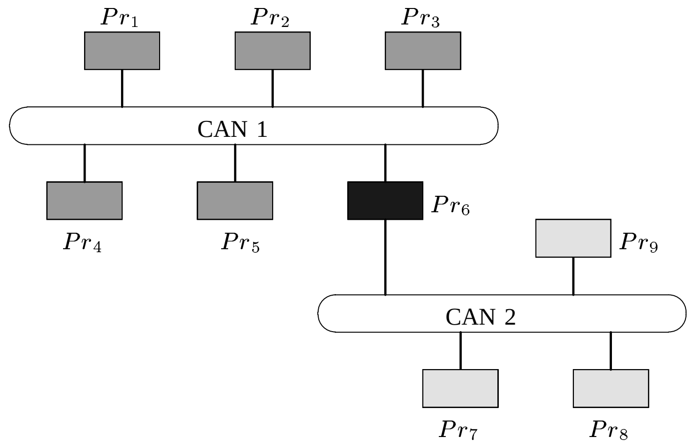

Fig. 1. A real-time distributed systems with 3 pools of processors: $\{Pr1, \ldots, Pr5\}, \{Pr6\}, \{Pr7, \ldots, Pr9\}$

B. Hardware architecture

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

Definition 2: 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.

Currently supported networks are based on fixed-priority to schedule frames on the communication medium, as in CAN. Others networks could be considered like TTCAN, TTP/C, etc. Figure 1 presents a supported distributed architecture.

III. A Branch and Bound method

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 an holistic analysis allows to conclude if this corresponding solution is feasible or not. To limit the combinatorial explosion while enumerating scheduling decisions, evaluations are performed to prune nodes that do not lead to feasible schedules.

We enumerate pool one by one and within a pool tasks are enumerated one by one. At each stage, the current task is allocated to a processor and it is assigned to a priority level.

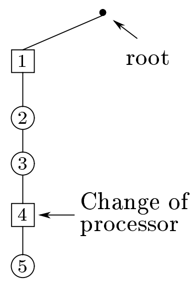

Fig. 2. Allocation and priority assignment in path of the search tree

Tasks τ_i	Pr_p	π_i
1	1	0
2	1	1
3	1	2
4	2	0
5	2	1

TABLE I
ALLOCATIONS AND PRIORITY ASSIGNEMENTS OF FIGURE 2

Definition 3: For a partial solution k of the scheduling problem, $G(k)$ is the communication graph associated to k and $\tau(k)$ is the current task of the node k.

The transformation of the communication graph associated to current vertex k^* of the search tree is defined below:

- if all receivers of m are allocated to the current processor, then the message m and all its adjacent edges are deleted .
- otherwise, the edge between m and τ_i is deleted.

A. The search tree

We extend the principles presented in [16] to enumerate without redundancy allocations and priority assignments. We first detail the enumeration of one pool; the complete search tree will be described at the end of this section. Two kinds of vertices are defined:

- 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.

The enumeration principle is presented in Figure 2 and the corresponding allocations and priority assignments are presented in Table I.

The search tree corresponding to a given pool is defined by the following rules:

1) a fictitious node defines the root of the search tree.
2) the first level is defined by $n - m + 1$ square nodes, where n is the number of tasks to allocate and m is the number of processors in the pool.
3) A path from the root to a node in the level i cannot be extended by a square node or a circle node if the rules 4,5,6 and 7 are not satisfied.
4) Every task is enumerated once.
5) A square node k cannot extend the current path if there exists a square node l, such that $l > k$, and belonging to the path joining the root and k.
6) Every path from the root to a leaf contains $\min(n, m)$ square nodes (i.e. every processor is used).
7) If a task k receives a message from a task i, both allocated to the current processor, then k cannot be inserted in the search tree. This rule ensures that a sender always has a higher priority than a receiver when they are allocated to the same processor. As a consequence, the precedence relation between senders and receivers is enforced.

The same principle is used to enumerate remaining pools. Lastly, priority assignment for messages are performed. The corresponding subtrees contain only circle nodes, modeling the priority assignment to messages. All subtrees are joined by fictitious roots. The complete structure of the search tree is presented in Figure 3.

B. Branching rules

Several branching rules have been implemented:

- pools are sorted in non-decreasing order of their weighted workloads: $\frac{1}{m_i}\sum_{k=1}^{n_i}\frac{C_k}{T_k}$.
- tasks are sorted in non decreasing order of their deadlines. Thus, the first enumerated path corresponds to the Deadline Monotonic policy for each processor.
- 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. 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 round-robin policy. Furthermore, such an approach can be more beneficial if a parallel computer is used to run the method.

C. Evaluation

To every leaf of the search tree, an holistic analysis is executed. The holistic analysis computes the worst-case response time of tasks and messages subjected to release jitters. 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:

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

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.

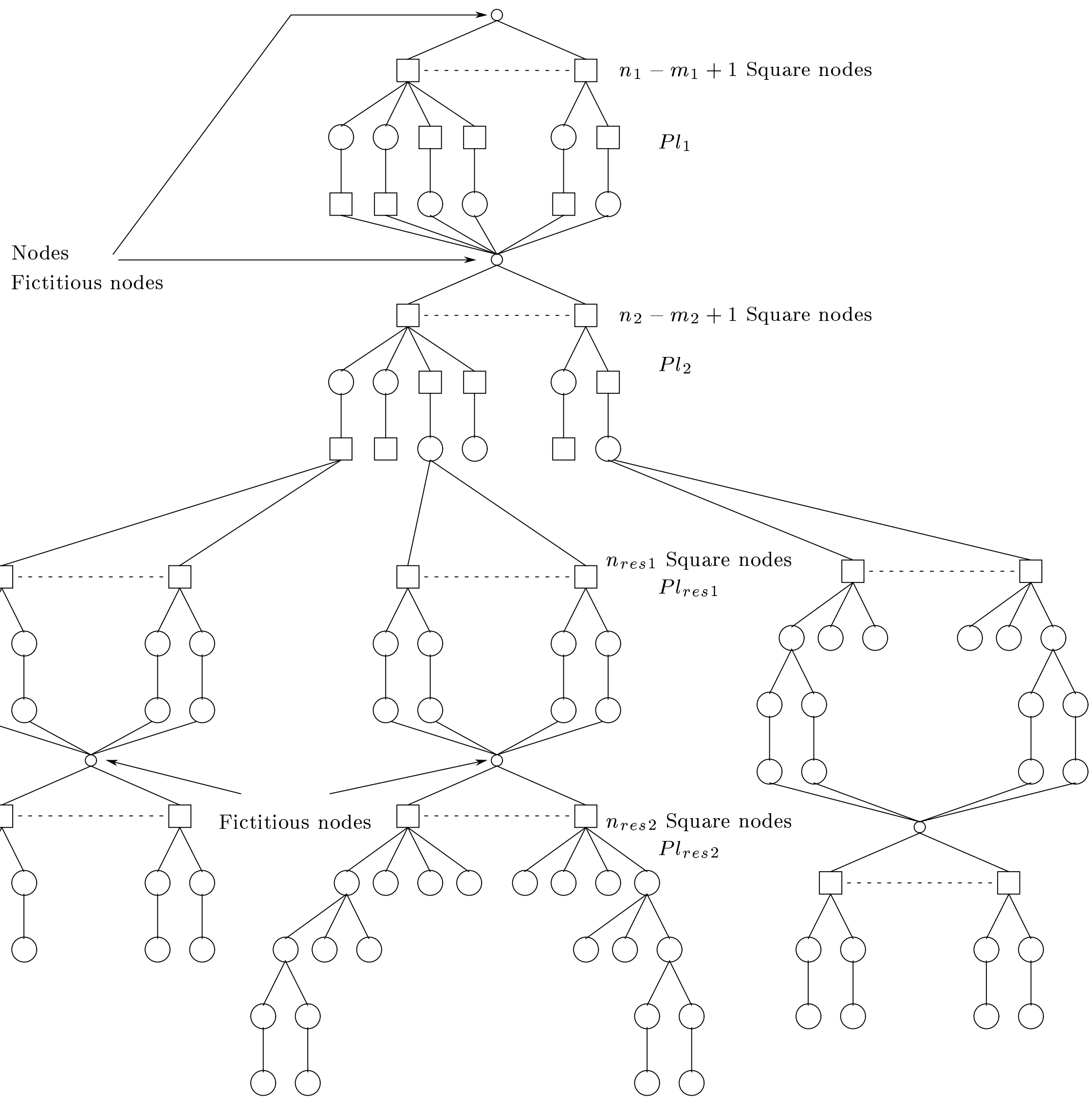

Fig. 3. General structure of the search tree

1) Updating release jitters: The function $Propag$ updates release jitters of tasks and messages. If a task has no input communication, then it has no release jitter (i.e., $J_i = 0$).

- *a task* τ_i *:* if $\{^\bullet i\} = \phi$, then $LB(J_i) = 0$. Otherwise, the lower bound of the release jitter is obtained by:

$$LB(J_i) = \max_{k \in \{^\bullet i\}} \{LB(Tr_k)\} \qquad \forall i \in S$$

- *a message* m_i *:* when a message m_i is considered, then receiver and sender of the message m_i are not allocated to the same processor. Thus, there exists a task $\tau_k \in \{^\bullet i\}$. The release jitter associated to message m_i is equal to the worst-case response time of τ_k.

$$LB(J_i) = Tr_k$$

2) Lower bounds: The function $Eval$ computes lower bounds (LB) of worst-case response times of tasks or messages. From a practical point of view, a message can be viewed as a task scheduled upon a non-preemptive processor (i.e. a network). Both cases will be next considered.

Calculting worst-case response time is a classical problem in the literature [17], [4]. When fixed-priority schedulers are considered, the worst-case response time of a task (or a message) assigned to the priority level i is obtained in an interval of time in which the processor runs tasks having a priority higher or equal to i. Such an interval of time, is called a i*-level Busy Period* [4]. The longest busy period is obtained when tasks are synchronously released at the beginning of the busy period (i.e., a critical instant [17]). We extend these classical results to compute lower bounds of worst-case response times of tasks and messages. Next, we assume that $[i]$ is the index of the task assigned to the i^{th} priority level.

Calculating $Tr_{[i]}$ consists in examinating the instance of $\tau_{[i]}$ executed in a i-level busy period. If the task has been assigned a priority by the Branch and Bound algorithm, then the function $Eval$ is defined by:

$$\begin{aligned} Int^{(k+1)} &= C_{[i]} + \sum_{j=0}^{i-1} \left\lceil \frac{J_{[j]} + Int^{(k)}}{T_{[j]}} \right\rceil C_{[j]} \\ LB(Tr_{[i]}) &= Int + LB\left(J_{[i]}\right) \end{aligned}$$

Task: The function Int, that calculates the workload of higher priority tasks, depends on tasks having higher priorities than τ_i. Two cases have to be considered according to prioritized or unprioritized tasks (i.e. tasks that have not be considered in the search tree).

- if the task τ_i has been allocated and has a priority l, then higher priority tasks are also known. A lower bound of the worst-case response time can be computed by:

$$\begin{aligned} Int^{(k+1)} &= C_i \\ &+ \sum_{j=0}^{l-1} \left\lceil \frac{LB\left(J_{[j]}\right) + Int^{(k)}}{T_{[j]}} \right\rceil C_{[j]} \\ LB\left(Tr_i\right) &= Int + LB\left(J_i\right) \end{aligned}$$

  The fixed point is reached for the smallest integer k such that:

$$Int = Int^{(k+1)} = Int^{(k)}$$

- If task τ_i is unprioritized, we separately study two cases depending on the index of the current processor. Let Pr_c be the current processor in the enumerated pool Pl_c.
  - If Pr_c is the last processor of Pl_c and if $\tau_i \in \Theta_c$, then tasks allocated to Pl_c have a higher priority than the evaluated task. Then, a subset of tasks having higher priorities than τ_i is known. As a consequence, a lower bound of the worst-case response time of τ_i is calculated by:

$$\begin{aligned} Int^{(k+1)} &= \sum_{j=0}^{l} \left\lceil \frac{LB(J_{[j]}) + Int^{(k)}}{T_{[j]}} \right\rceil C_{[j]} \\ &+ C_i \end{aligned}$$

    The fixed point is defined by the smallest integer k such that:

$$Int = Int^{(k+1)} = Int^{(k)}$$

    A lower bound of the worst-case response time of τ_i is calculated by:

$$LB\left(Tr_i\right) = Int + LB\left(J_i\right)$$

  - If Pr_c is not the last processor of Pl_c or if $\tau_i \notin \Theta_c$, then no information is known about task having a higher priority than τ_i (e.g. τ_i can be assigned to the highest priority level on the next processor). Thus, a lower bound of the worst-case response time of τ_i is defined by:

$$LB(Tr_i) = LB(J_i) + C_i$$

Message: results obtained for preemtive tasks can be easily extended to non-preemptive dispatching strategies. The longest busy period is not necessarily started by a critical instant. When preemption is not allowed, the critical instant of a i-level busy period can be postponed by the longest task having a priority lower than i. Such a delay is called a *blocking time* in the literature. Thus, the worst-case delay occurs when a lower priority task is begun just before a critical instant (at time 0 minus ϵ, where ϵ is an arbitrary small number). The Lower bound of worst-case response times of messages is defined by the lower bound of the release jitter plus a lower bound of the worst-case interference due to higher priority messages and the non-preemptive dispatching strategy. We also have to consider two cases according to the status of the message: prioritized or not.

- If the message m_i is prioritized: all tasks have been allocated and prioritized. The set of messages is given by the communication graph associated to the currently explored vertex in the search tree. Furthermore, messages having a higher priority than m_i are also known since priorities are allocated in non-decresing order of the priority levels. Let $\Theta_r^\star$ be the set of unprioritized message. A lower bound of the blocking time is obtained while considering the longest task in the set $\Theta_r^\star$. The worst-case interference for the evaluated message is defined by:

$$\begin{aligned} Int^{(k+1)} &= C_i \\ &+ \sum_{j=0}^{i-1} \left(\left\lfloor \frac{LB(J_{[j]}) + Int^{(k)}}{T_{[j]}} \right\rfloor + 1 \right) C_{[j]} \\ &+ \max_{k \in \Theta_r^\star}(C_k) \end{aligned}$$

- if the message m_i is unprioritized: let Pl_r be the network associated to m_i:
  - If $Pl_c = Pl_r$ then all tasks have been prioritized. If the evaluated message has not yet be prioritized, then it can be scheduled with a lower priority level. As a consequence the only possible lower bound on the blocking time is 0. Otherwise if it is prioritized, the worst-case interference supported by m_i is defined by (assuming that it is assigned to priority level l):

$$\begin{aligned} &Int^{(k+1)} = C_i \\ &+ \sum_{j=0}^{l} \left(\left\lfloor \frac{LB(J_{[j]}) + Int^{(k)}}{T_{[j]}} \right\rfloor + 1 \right) C_{[j]} \end{aligned}$$

  - If $Pl_c \neq Pl_r$ then all tasks have not been scheduled. Since scheduling decisions for messages are taken when all scheduling decisions have been taken for tasks (in order to determined exactly the communcation graph) then a lower bound of the worst-case response time of the evaluated message is obtained by considering that it is assigned the highest priority level. Thus, we obtain:

$$Int = C_i \tag{1}$$

Lastly, if all tasks and messages have been allocated and prioritized then our evaluation process is exactly an holistic analysis.

D. Elimination rule

Lower bounds previously defined are used to prune the current vertex k^* in the search tree. For that purpose we defined the following rule.

Theorem 1: If there exists $i \in V$ in $G(k^*) = (V, E)$ such that $LB(Tr_i) > D_i$ then the current allocation and priority assignment cannot lead to a feasible schedule. As a consequence, no child of the current vertex will be considered and a backtracking is operated.

Proof: We only detail the proof sketch. Let k and k' be two vertices in the search tree such that $k \prec k'$, the proof is defined by two stages :

- The communication graphs of k end k' satisfy the following property: $G(k') \subseteq G(k)$. This is a direct consequence of the transformation process of the communication graph.
- Lower bounds of worst-case response times of tasks and messages are non-decreasing in every path starting from the root and ending by any vertex. This is a direct property of our evaluation process based on the principles of the holistic analysis.

As a consequence, extending a path in the search tree such that $LB(Tr_i) > D_i$ for some task or message $i \in V$ cannot lead to a feasible schedules since $Tr_i \geq LB(Tr_i)$ for all $i \in V$. $\square$

The Branch and Bound completes when a leaf leading to a feasible holistic schedule is reached. Implementation details are presented in [18].

IV. Numerical experimentations

We detail two kinds of numerical experimentations:

- Randomly generated set of tasks.
- A real-size application corresponding to the architecture presented in Figure 1.

A. Randomly generated configurations

In order to evaluate the capabilities to allocate tasks, we randomly generate set of tasks to be run upon a multiprocessor system. There is one pool and one network (CAN). For a fixed number of tasks and processors, we generate 50 instances. A time limit has been set to one hour. Thus, if no feasible has been found within the time limit, then the configuration is unvalidated. Our experimentations have shown that if workloads of processors are high or low, then our method fastly reaches a feasible schedule or proves that there is no feasible holistic schedule. The longest execution times of the method are obtained when processors are loaded at a level closed to 50 percent.

Figure 4 gives the mean execution time of the method (within the time limit) for configurations having workloads belonging to: $[40, 50)$, $[50, 60)$ and $[60, 70)$. The holistic analysis is often viewed as a very pessimistic analytic tool. Our experimentations shows that instances of problems can have processor with a worload of 70 percents (this is a huge workload for a hard real-time system).

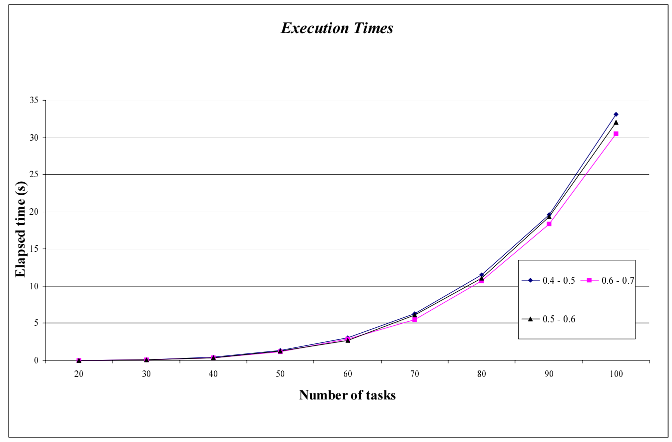

Fig. 4. Mean resolution times for multiprocessor architectures

Figure 5 gives the number of validated configutations in function of the number of processors, messages and tasks. In all these experimentations, the workload of every pools of processors belongs to the interval $[50, 60)$ percents.

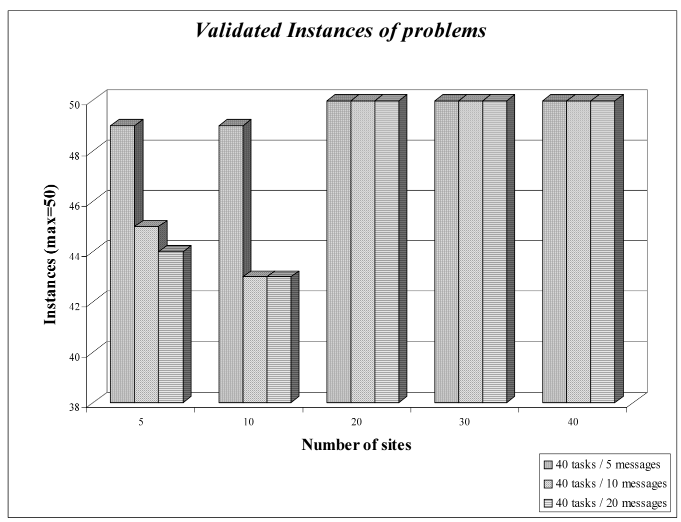

Fig. 5. Number of validated configurations

B. A real-size system

We consider the system presented in Figure 1. 24 tasks have to be allocated upon 5 indentical processors of the first pool. The second pool consists in a single processor and has to run 7 tasks. Precedence relations are given in Figure 6. Lastly, the third pool has 3 processors and 13 tasks. The first CAN has to

Pl	Task	C_i	D_i	Pl	Task	C_i	D_i
Pool 1 (Proc 1, 2, 3, 4 and 5)	τ_0	1.0	10	Pool 2 (Proc6)	τ_{24}	2.0	50
	τ_1	2.0	20		τ_{25}	2.0	50
	τ_2	2.0	100		τ_{26}	2.0	10
	τ_3	2.0	15		τ_{27}	2.0	100
	τ_4	2.0	14		τ_{28}	2.0	40
	τ_5	2.0	50		τ_{29}	2.0	20
	τ_6	2.0	40		τ_{30}	2.0	100
	τ_7	2.0	15	Pool 3 (Proc 7, 8 and 9)	τ_{31}	2.0	150
	τ_8	2.0	50		τ_{32}	2.0	200
	τ_9	2.0	50		τ_{33}	2.0	50
	τ_{10}	2.0	14		τ_{34}	2.0	150
	τ_{11}	2.0	20		τ_{35}	2.0	50
	τ_{12}	2.0	40		τ_{36}	2.0	50
	τ_{13}	2.0	15		τ_{37}	2.0	10
	τ_{14}	2.0	100		τ_{38}	2.0	100
	τ_{15}	2.0	20		τ_{39}	2.0	150
	τ_{16}	2.0	20		τ_{40}	2.0	100
	τ_{17}	4.0	14		τ_{41}	2.0	150
	τ_{18}	4.0	20		τ_{42}	2.0	200
	τ_{19}	2.0	20		τ_{43}	2.0	50
	τ_{20}	2.0	20	Pool 4 (CAN 1)	$M1$	0.51	10
	τ_{21}	2.0	10		$M2$	0.32	14
	τ_{22}	2.0	14		$M3$	0.32	20
	τ_{23}	2.0	15		$M4$	0.29	15
Pool 5 (CAN 2)	$M13$	0.2	50		$M5$	0.39	20
	$M14$	0.5	10		$M6$	0.39	40
	$M15$	0.1	150		$M7$	0.36	15
	$M16$	0.5	200		$M8$	0.39	50
	$M17$	0.2	100		$M9$	0.36	20
	$M18$	0.4	150		$M10$	0.47	100
	$M19$	0.4	150		$M11$	0.39	50
					$M12$	0.25	100

TABLE II
REAL-TIME SOFTWARE ARCHITECTURE

transport 12 messages and the second one 7 messages. Table II summaries the software architecture, parameters of tasks and messages.

The Table III gives the results of the Branch and Bound. A bullet means that the message has not been sent upon a network but is associated to a local communication within the local memory of a processor (sender and receiver are allocated to the same processor). The running time of our algorithm for the presented case study is 182 seconds upon a standard PC[1].

V. CONCLUSION

We have presented a Branch and Bound method that automatically allocates tasks to processors and assigns fixed-priority to tasks. The main contributions in this paper are the following: to simultaneously allocate tasks and assign their priorities and to use the principles of the holistic analysis to calculate lower bounds of worst-case response times for tasks and messages. Numerical experimentations show that the method can find feasible schedules even if the workdload of the system is high and the method is applicable for real-size application. Other computational results are presented in [18].

[1]Pentium IV, 512 Mo RAM.

S	τ_i	π_i	Tr_i	J_i	S	τ_i	π_i	Tr_i	J_i	S	τ_i	π_i	Tr_i	J_i	S	τ_i	π_i	Tr_i	J_i
Proc 1	τ_7	0	2.0	0.0	Proc 4	τ_{20}	1	10.37	6.37	Proc 7	τ_{33}	1	4.0	0.0	CAN 1	M_6	4	4.41	2.0
	τ_{17}	1	6.0	0.0		τ_6	2	10.41	4.41		τ_{36}	2	34.69	26.69		M_8	5	10.80	8.0
	τ_{18}	2	14.02	4.02		τ_5	3	18.80	10.80		τ_{43}	3	6.00	0.00		M_{11}	6	13.19	10.0
	τ_4	3	12.0	0.0		τ_8	4	10.0	0.0		τ_{38}	4	17.70	5.70		M_{10}	7	9.44	6.00
	τ_{10}	4	14.00	0.0	Proc 5	τ_{12}	0	2.0	0.0	Proc 8	τ_{35}	0	2.0	0.0		M_{12}	8	11.44	8.0
Proc 2	τ_{13}	0	2.0	0.0		τ_9	1	14.80	10.80		τ_{40}	1	10.0	0.0		M_2	•	•	•
	τ_{21}	1	5.98	1.98		τ_2	2	6.00	0.0		τ_{31}	2	6.0	0.0		M_3	•	•	•
	τ_{22}	2	6.0	0.0		τ_{14}	3	8.00	0.0		τ_{34}	3	11.70	3.70.0		M_7	•	•	•
	τ_3	3	11.66	3.66	Proc 6	τ_{26}	0	3.98	1.98		τ_{39}	4	17.70	5.7	CAN 2	M_{14}	0	4.88	3.98
	τ_{23}	4	14.00	0.0		τ_{29}	1	8.02	4.02	Proc 9	τ_{41}	0	2.0	0.0		M_{15}	1	3.30	2.0
Proc 3	τ_0	0	1.0	0.0		τ_{28}	2	10.41	4.41		τ_{32}	1	4.0	0.0		M_{13}	2	26.69	25.19
	τ_1	1	3.0	0.0		τ_{24}	3	8.0	0.0		τ_{42}	2	6.0	0.0		M_{19}	3	3.70	2.0
	τ_{11}	2	5.0	0.0		τ_{25}	4	25.19	13.19	CAN 1	M_1	0	1.98	1.0		M_{18}	4	5.70	4.0
	τ_{15}	3	7.0	0.0		τ_{27}	5	23.44	9.44		M_5	1	6.37	5.0		M_{16}	•	•	•
	τ_{16}	4	13.02	4.02		τ_{30}	6	29.44	11.44		M_4	2	3.66	2.0		M_{17}	•	•	•
	τ_{19}	0	2.0	0.0		τ_{37}	0	6.88	4.88		M_9	3	4.02	2.0					

TABLE III
ALLOCATION AND PRIORITY ASSIGNMENT RESULTS

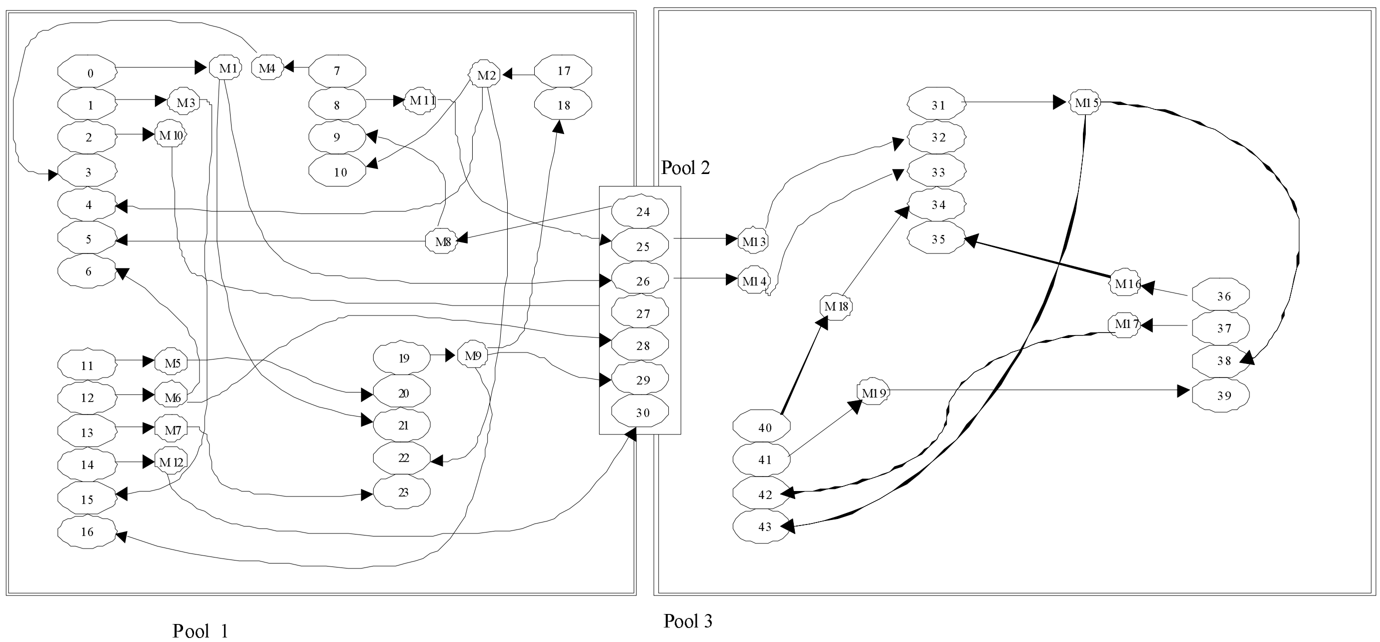

Fig. 6. Precedence relations among tasks and messages

The actual version of the method is mainly useful to validate applications. In further works we want to take into account more practical factors such as allocation constraints and size of available memory to task allocations, and also economical factors such as minimizing the number of required processors, etc. These extensions should be very helpful at the design step of a real-time distributed system.

REFERENCES

[1] J. Thomesse, “Fieldbuses and interoperability,” *Control Engineering Practice*, vol. 7, pp. 81–94, 1999.

[2] M. Richard, P. Richard, and F. Cottet, “Task and message priority assignment in automotive systems,” in 4^{th} *FeT IFAC Conference on Fieldbus Systems and their Applications*, 15,16 November 2001, pp. 105–112.

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

[4] J. P. Lehoczky, “Fixed priority scheduling of periodic task sets with arbitrary deadlines,” in *Proceedings of Real-Time Systems Symposium*, 1991, pp. 166–171.

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

[6] 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*, vol. 13, no. 2, pp. 167–199, 1997.

[7] 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)*, 1997, pp. 33–38.

[8] P. Altenbernd and H. Hansson, “The slack method: A new method for static allocation of hard real-time tasks,” *Journal of Real-Time Systems*, vol. 13, no. 2, pp. 103–130, Septembre 97.

[9] K. Tindell, A. Burns, and A. Wellings, “Allocating hard real-time tasks (an np-hard problem made easy),” *Real-Time Systems*, vol. 4, no. 2, pp. 145–165, 1992.

[10] 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.

[11] Y. OH and S. H. Son, “Allocating fixed-priority periodic tasks on multiprocessor systems,” *Real-Time System Journal*, vol. 9, no. 3, pp. 207–239, 1995.

[12] 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*, vol. 23, no. 12, 1997.

[13] 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*. Montreal, Canada, 21-25 October 1996, pp. 226–229.

[14] K. W. Tindell, “Fixed priority scheduling of hard real-time systems,” Ph.D. dissertation, Univeristy of York, 1994.

[15] K. W. Tindell and J. Clark, “Holistic schedulability analysis for distributed hard real-time systems,” *Microprocessors and Microprogramming*, vol. 40, 1994.

[16] P. Bratley, M. Florian, and P. Robillard, “Scheduling with earliest start and due date constraints on multiple machines,” *Naval Research Logistic Quaterly*, vol. 22, no. 1, pp. 165–173, 1975.

[17] J. C. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in hard real-time evironment,” *ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[18] M. Richard, *Contribution à la validation des systèmes temps réel distribués : ordonnancement à priorités fixes et placement*. PhD Thesis, University of Poitiers, 2002.