Allocating and Scheduling Tasks in Multiple Fieldbus Real-Time Systems
Michael Richard, Pascal Richard, Francis Cottet

To cite this version:
 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 proving 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 instance, in automotive applications, a widely 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. 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 has a worst-case execution 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
\( \tau_i \) is schedulable if its worst-case response time \( T_{R_i} \) 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 \( \tau_i \) is denoted \( \pi_i \); 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 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 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 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 \( \tau_i \); if \( \tau_i \) is not yet allocated, then we note \( P(i) = \emptyset \). 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 immediate 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:

\[
\begin{align*}
S \cap M &= \emptyset \\
E &\subseteq (S \times M) \cup (M \times S) \\
\forall i \in M, \quad |i\bullet| &= 1 \\
\forall i \in M, \forall k \in \{i\bullet\} \quad &\text{if } P(i\bullet) \neq \emptyset \text{ and } P(k) \neq \emptyset \\
\text{then } \quad P(i\bullet) &\neq P(k)
\end{align*}
\]

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.

### 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. 1. A real-time distributed systems with 3 pools of processors: \{Pr1,...,Pr5\}, \{Pr6\}, \{Pr7,...,Pr9\}](image-url)
and 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: \( \frac{1}{m_i} \sum_{k=1}^{n_i} C_k \).
- 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:

\[
\forall i \in V \quad \begin{cases} 
    LB(T_{r_i}^{(k)}) = Eval\left(LB\left(j_i^{(k-1)}\right)\right) \\
    LB(j_i^{(k)}) = Propag\left(LB\left(T_{r_i}^{(k)}\right)\right)
\end{cases}
\]

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

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

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.

![Image](image1.png)

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

<table>
<thead>
<tr>
<th>Tasks</th>
<th>( \tau_i )</th>
<th>( P_{T_p} )</th>
<th>( \sigma_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 1**

**Allocations and Priority Assignments of Figure 2**
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 $\tau_i$: if $\{\bullet i\} = \emptyset$, then $LB(J_i) = 0$. Otherwise, the lower bound of the release jitter is obtained by:
  $$LB(J_i) = \max_{k \in \{\bullet i\}} \{LB(Tr_k)\} \quad \forall i \in S$$

- 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 $\tau_k$.
  $$LB(J_i) = Tr_k$$

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.

Calculating 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 \( T_{r_i} \) consists in examining 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 \( \text{Eval} \) is defined by:

\[
\text{Int}^{(k+1)} = C_i + \sum_{j=0}^{i-1} \left( \frac{\text{LB}(J_{[j]}) + \text{Int}^{(k)}}{T_{[j]}} \right) C_{[j]}
\]

\[
\text{LB}\left( T_{r_i} \right) = \text{Int} + \text{LB}\left( J_i \right)
\]

**Task:** The function \( \text{Int} \), that calculates the workload of higher priority tasks, depends on tasks having higher priorities than \( \tau_i \). 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 \( \tau_i \) 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:

\[
\text{Int}^{(k+1)} = C_i + \sum_{j=0}^{i-1} \left( \frac{\text{LB}(J_{[j]}) + \text{Int}^{(k)}}{T_{[j]}} \right) C_{[j]}
\]

\[
\text{LB}\left( T_{r_i} \right) = \text{Int} + \text{LB}\left( J_i \right)
\]

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

\[
\text{Int} = \text{Int}^{(k+1)} = \text{Int}^{(k)}
\]

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

\[
\text{Int}^{(k+1)} = \sum_{j=0}^{i} \left( \frac{\text{LB}(J_{[j]}) + \text{Int}^{(k)}}{T_{[j]}} \right) C_{[j]}
\]

\[
\text{LB}\left( T_{r_i} \right) = \text{Int} + \text{LB}\left( J_i \right)
\]

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

\[
\text{Int} = \text{Int}^{(k+1)} = \text{Int}^{(k)}
\]

A lower bound of the worst-case response time of \( \tau_i \) is calculated by:

\[
\text{LB}\left( T_{r_i} \right) = \text{Int} + \text{LB}\left( J_i \right)
\]

- If \( P_{r_c} \) is not the last processor of \( P_{l_c} \), then no information is known about task having a higher priority than \( \tau_i \) (e.g. \( \tau_i \) can be assigned to the highest priority level on the next processor). Thus, a lower bound of the worst-case response time of \( \tau_i \) is defined by:

\[
\text{LB}(T_{r_i}) = \text{LB}(J_i) + C_i
\]

**Message:** results obtained for preemptive 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 \( \epsilon \), where \( \epsilon \) 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-decreasing order of the priority levels. Let \( \Theta^*_i \) be the set of unprioritized message. A lower bound of the blocking time is obtained while considering the longest task in the set \( \Theta^*_i \). The worst-case interference for the evaluated message is defined by:

\[
\text{Int}^{(k+1)} = C_i + \sum_{j=0}^{i-1} \left( \frac{\text{LB}(J_{[j]}) + \text{Int}^{(k)}}{T_{[j]}} \right) + 1 \right) C_{[j]}
\]

\[
\text{LB}\left( T_{r_i} \right) = \text{Int} + \text{LB}\left( J_i \right)
\]

- If the message \( m_i \) is unprioritized: let \( P_{r_c} \) be the network associated to \( m_i \):
  - If \( P_{r_c} = P_{l_c} \) 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 \( l \)):

\[
\text{Int}^{(k+1)} = C_i + \sum_{j=0}^{i-1} \left( \frac{\text{LB}(J_{[j]}) + \text{Int}^{(k)}}{T_{[j]}} \right) + 1 \right) C_{[j]}
\]

- If \( P_{r_c} \neq P_{l_c} \) 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 communication 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:

\[ \text{Int} = C_i \]  \hspace{1cm} (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 \( k \) and \( k' \) 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 workload of 70 percents (this is a huge workload for a hard real-time system).

![Fig. 4. Mean resolution times for multiprocessor architectures](image)

Figure 5 gives the number of validated configurations 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](image)

B. A real-size system

We consider the system presented in Figure 1. 24 tasks have to be allocated upon 5 indetical 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
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 workload 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.
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


