Combined Task and Message Scheduling in Distributed Real-Time Systems

Tarek F. Abdelzaher, Member, IEEE Computer Society, and Kang G. Shin, Fellow, IEEE

Abstract—This paper presents an algorithm for off-line scheduling of communicating tasks with precedence and exclusion constraints in distributed hard real-time systems. Tasks are assumed to communicate via message passing based on a time-bounded communication paradigm, such as the real-time channel [1]. The algorithm uses a branch-and-bound (B&B) technique to search for a task schedule by minimizing maximum task lateness (defined as the difference between task completion time and task deadline), and exploits the interplay between task and message scheduling to improve the quality of solution. It generates a complete schedule at each vertex in the search tree, and can be made to yield a feasible schedule (found before reaching an optimal solution), or proceed until an optimal task schedule is found. We have conducted an extensive simulation study to evaluate the performance of the proposed algorithm. The algorithm is shown to scale well with respect to system size and degree of intertask interactions. It also offers good performance for workloads with a wide range of CPU utilizations and application concurrency. For larger systems and higher loads, we introduce a greedy heuristic that is faster but has no optimality properties. We have also extended the algorithm to a more general resource-constraint model, thus widening its application domain.

Index Terms—Real-time scheduling, combined task and message scheduling, distributed hard real-time systems, resource constraints, deadlock.

1 INTRODUCTION

In hard real-time systems, failure to meet the deadline of a task may result in catastrophic consequences. Each task must therefore be guaranteed a priori to meet its timing constraint, and hence, efficient techniques for preruntime scheduling or schedulability analysis are needed. Since periodic tasks are the base load of such systems, we shall focus on how to schedule them. Sporadic tasks may be considered periodic by using, for example, the sporadic server. For the algorithm to be of practical value, task precedence constraints and resource requirements need to be taken into account. Furthermore, the algorithm should find a feasible schedule (i.e., one that meets all deadlines), whenever such a schedule exists. One approach is to cast this problem into that of minimizing maximum task lateness. A feasible schedule would then correspond to a solution where maximum task lateness is nonpositive. If the optimal schedule has positive lateness, then we know that no feasible schedule exists. Thus, the problem is to find the optimal\(^1\) preruntime schedule for hard real-time tasks with known arrival times, precedence constraints, and resource requirements. Such a problem was solved by Xu and Parnas [2] for uniprocessor systems. An attempt to extend their approach to several processors was made by Shepard and Gagne [3], but their algorithm occasionally fails to find existing feasible schedules, as we pointed out in [4]. Xu [5] remedied this shortcoming and optimally solved the problem for multiprocessor systems. However, his model is not suitable for distributed systems, since it assumes that tasks can be resumed on any processor at no additional cost, neglects the cost of intertask communication, and does not address the problem of scheduling intermachine messages.

In distributed hard real-time systems, intermachine message communication affects task schedulability, and thus, has to be taken into account. One way to solve this problem is to separate message communication from task scheduling. A communication paradigm such as the real-time channel [1] is first assumed where the communication subsystem guarantees bounded message delays (specified as message deadlines), then the task scheduling problem is solved on top of that paradigm, assuming fixed and known message delay bounds. An optimal algorithm for solving the latter problem is presented in [6], but it does not consider resource requirements. In general, a disadvantage of separating message scheduling from task scheduling is that the bounded message delays guaranteed by solving the former are a function of the specified message deadlines. However, message deadlines cannot be accurately computed before a task schedule is computed (from which, for example, we may want to know the time to send these messages, and the lateness of their receivers), but the task schedule cannot be computed without an assumption about message delay bounds in the first place. Because of this tight coupling between the end-to-end message delays achievable for different message deadline assignments, as well as task lateness in the corresponding task schedules,\(^2\) the problems of task scheduling and message scheduling

---

1. We mean this in the sense of minimizing maximum task lateness.

2. Actually, there exists a circular dependency.
should be solved together. We must therefore develop a combined approach to schedulability analysis that takes into consideration both tasks and intertask messages. Several heuristics have been proposed to solve the combined problem, e.g., [7] and [8]. A flexible scheme which combines off-line analysis with on-line guarantees is suggested in [9] for uniprocessors. In [10], a rather similar scheme is described for distributed systems. It uses off-line analysis to convert task precedence and communication constraints into pseudodeadlines of tasks and messages, then employs an on-line guarantee routine to find a runtime task and message schedule that minimizes the number of tasks missing deadlines. An algorithm combining this problem with that of task allocation was presented in [11].

In contrast, we propose an optimal algorithm for scheduling tasks in a distributed real-time system which interacts with the problem of message scheduling, thereby improving the quality of solution. Assuming that the real-time channel paradigm [1] is used for message communication, the problem of message scheduling is reduced to that of an appropriate choice of message deadlines. Let’s define the message-priority space in which each point corresponds to a different message-priority assignment. Our algorithm may be viewed as searching the message-priority space and the space of all task schedules for a point where the task scheduling problem has an optimal solution. Conceptually, the search proceeds in two orthogonal dimensions. The first searches the message priority space. At a given point in the message-priority space, the second searches the space of all possible task schedules for a schedule that minimizes maximum task lateness. Due to the complexity of the combined problem, optimality is guaranteed in the second dimension, while a near optimal solution is sought in the first dimension.

Since it often suffices in hard real-time systems to find a feasible schedule which satisfies all deadlines as opposed to an optimal one, the algorithm can be terminated after finding a first feasible schedule. This does not eliminate the need for an optimal approach, since in case the best schedule found so far is not feasible, we should be able to tell whether there are no feasible solutions to the problem.

The rest of this paper is organized as follows: Section 2 presents the basic algorithm. Section 3 evaluates its performance. A fast heuristic derived from the algorithm is presented in Section 4, which uses a greedy technique (instead of optimal) to search for feasible schedules. The basic algorithm is generalized to a more practical model for resource requirements in Section 5. The paper concludes with Section 6.

2 The Basic Algorithm

This section presents the basic algorithm proposed for combined task and message scheduling. Section 2.1 describes the system model and notation, while Section 2.2 presents a general solution approach. Simulation results are provided in Section 3.

2.1 System Model

The distributed system is composed of a set, PN, of p processing nodes (PNs), PN1, · · · , PNp, connected by an arbitrary network N. PNs run a set T of n hard real-time tasks, T1, · · · , Tn. Each task is assumed to reside permanently on one processor. How to assign tasks to processors is beyond the scope of this paper. See [12] for a suitable task assignment algorithm. Each task Tk in the distributed system has a known arrival time ak, total execution time ek, and deadline dk. In the case of periodic tasks, each task also has a period Pk. The arrival time of periodic task Tk is associated with its individual invocations, such that the arrival time of its jth invocation, Tk[j], is the beginning of its period ak = (j − 1)Pk. The deadline of a periodic task invocation is set relative to its arrival time. For periodic tasks it suffices to analyze the system within an interval of time equal to the least common multiple (LCM) of all task periods. We call this interval the planning cycle. Otherwise, the planning cycle is the duration of the entire off-line schedule. In this paper we focus on periodic tasks. Table 1 gives an example set of periodic tasks assigned to two PNs. The duration of the planning cycle is LCM(3, 6, 12) = 12. This set will be used throughout the rest of the paper to illustrate the solution approach.

A task may be composed of one or more modules. Each module Mi of a task invocation Tk[j] has the worst-case execution time Ci, which bounds its actual execution time, the arrival time Ai, which denotes the earliest time the module can be invoked, and the deadline Di, which is the latest time it can finish execution. Initially, Ai = ak[j], and Di = dk[j] of the corresponding task invocation. In a particular schedule γ, the time S(γ) when module Mi is first given the CPU is called the module start time and the time E(γ) when the module finishes execution is called the module completion time. The completion time of a task invocation Tk[j] is the completion time Elast of its last module Mlast. The lateness of task invocation Tk[j] in schedule γ is defined as the lateness of the task’s last module, Elast(γ) − Dlast. A positive task lateness indicates that the task missed its deadline. Schedule lateness, laten(γ), is the maximum lateness over all task invocations in the schedule.

Modules may have synchronization constraints, which are either precedence constraints (e.g., when one module waits for the results of another) or mutual exclusion constraints (between pairs of modules accessing the same serial

<table>
<thead>
<tr>
<th>Taskk</th>
<th>PN</th>
<th>c_k</th>
<th>d_k – a_k</th>
<th>P_k</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>T_2</td>
<td>1</td>
<td>2</td>
<td>5.5</td>
<td>6</td>
</tr>
<tr>
<td>T_3</td>
<td>1</td>
<td>3</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>T_4</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>T_5</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>T_6</td>
<td>2</td>
<td>0.5</td>
<td>3.5</td>
<td>6</td>
</tr>
</tbody>
</table>
resource). A precedence constraint $M_i$ precedes $M_j$ means that $M_i$ and $M_j$ must be scheduled such that $E_{i}(\gamma) \leq S_{j}(\gamma)$. A mutual exclusion constraint $M_i$ excludes $M_j$ means that neither of the two modules can have an execution interval between the other’s start time and completion time. In other words, the modules must be scheduled such that either $M_i$ precedes $M_j$ or $M_j$ precedes $M_i$. Note that excludes is commutative. The set $\text{Sync}_{init}$ is the set of all synchronization constraints defined initially for the system. In this section, we cast resource constraints as mutual exclusion constraints defined initially for the system. In this section, we cast resource constraints as mutual exclusion constraints defined initially for the system. In this section, we cast resource constraints as mutual exclusion constraints defined initially for the system.

For our running example, Table 2 depicts all task invocations in the task set shown in Table 1 within the planning cycle. For simplicity of illustration, all tasks except one ($T_3$) are chosen to consist of only one module. To illustrate synchronization constraints, we let task $T_3$ (module $M_1$) use some results computed in module $M_7$ of task $T_4$ that are sent to $T_5$ via a message $m_{7,11}$. We also let each odd-numbered invocation of task $T_3$ (i.e., module $M_9$) communicate a message to the fourth invocation of task $T_1$ (module $M_1$) in the same planning cycle. Furthermore, we let tasks $T_3$ and $T_5$ use the same serial resource, thus creating a mutual exclusion constraint between each invocation of $T_4$ (modules $M_9$ and $M_{10}$) and task $T_5$ (module $M_{11}$). The resulting set of synchronization constraints $\text{Sync}_{init}$ is shown in Table 2. This example task set will be used throughout the paper to illustrate the algorithm.

### 2.2 The Solution Approach

Our objective is to find an optimal task schedule and a near-optimal message priority assignment in the sense of minimizing schedule lateness (defined in Section 2.1) across all PNs. In doing so, we consider C1 a message communication paradigm, C2 an optimal task scheduling algorithm, and C3 a message priority assignment heuristic.

The message communication paradigm, C1, defines the mechanism used for message transport on the target system. The paradigm itself is not a contribution of our algorithm, but a parameter of the underlying system. It must guarantee bounded communication delay for messages. We compute

1. a message priority order using some heuristic C3 that attempts to reduce task lateness,
2. message delay bounds using paradigm C1, and
3. an optimal schedule using the optimal task scheduling algorithm C2 that minimizes task latency for given message delays.

The real-time channels presented in [1] guarantee bounded communication delays and thus will be used as an example for the paradigm C1. The general idea of real-time channels is to reserve resources in the network (e.g., on network routers) to guarantee bounded-time processing of a stream of messages specified by a given period, maximum message size, and worst-case jitter.

A B&B technique is used to minimize the lateness of the distributed communicating tasks. It can be viewed as a search, by implicit enumeration, through the entire valid solution space. A valid solution, $\gamma$, is a schedule with the following properties.

- Every module $M_i$ in $\gamma$ starts no earlier than its arrival time, i.e., $S_{i}(\gamma) \geq A_{i}$, and is given the CPU for a total of $C_{i}$ time units.
- All task synchronization (i.e., precedence and exclusion) constraints in the set $\text{Sync}_{init}$ are satisfied.3
- Message delays are computed using paradigm C1 and are accounted for in the schedule. That is, if module $M_i$ sends a message $m_{i,j}$ to module $M_j$, and the delay bound of $m_{i,j}$ computed by paradigm C1 is $d_{i,j}$, then $S_{i}(\gamma) \geq d_{i,j} + E_{i}(\gamma)$.

A valid solution $\gamma$ is feasible if it satisfies the additional constraint that every module $M_i$ in $\gamma$ finishes before its deadline (i.e., $E_{i}(\gamma) \leq D_{i}$). The B&B search can be viewed as traversing a search tree. The root vertex, $V_{root}$, of the search tree represents the space of all possible valid solutions.

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Module</th>
<th>$a_k$</th>
<th>$c_k$</th>
<th>$d_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1[1]$</td>
<td>$M_1$</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>$T_1[2]$</td>
<td>$M_2$</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>$T_1[3]$</td>
<td>$M_3$</td>
<td>6</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>$T_1[4]$</td>
<td>$M_4$</td>
<td>9</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>$T_2[1]$</td>
<td>$M_5$</td>
<td>0</td>
<td>2</td>
<td>5.5</td>
</tr>
<tr>
<td>$T_2[2]$</td>
<td>$M_6$</td>
<td>6</td>
<td>2</td>
<td>11.5</td>
</tr>
<tr>
<td>$T_3[1]$</td>
<td>$M_7$</td>
<td>0</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>$M_8$</td>
<td>0</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>$T_4[1]$</td>
<td>$M_9$</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>$T_4[2]$</td>
<td>$M_{10}$</td>
<td>6</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>$T_5[1]$</td>
<td>$M_{11}$</td>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>$T_6[1]$</td>
<td>$M_{12}$</td>
<td>0</td>
<td>0.5</td>
<td>3.5</td>
</tr>
<tr>
<td>$T_6[2]$</td>
<td>$M_{13}$</td>
<td>6</td>
<td>0.5</td>
<td>9.5</td>
</tr>
</tbody>
</table>

Messages: $m_{7,11}, m_{9,4}$

$\text{Sync}_{init} = \{ M_i \text{ precedes } M_{11}, M_7 \text{ precedes } M_6, M_9 \text{ precedes } M_4, M_6 \text{ excludes } M_{11}, M_{10} \text{ excludes } M_{13} \}$

3. We have defined in Section 2.1 what it means to satisfy the precedence and exclusion constraints.
Branching from vertex $V$ is a subdivision of the solution space of the parent among a set of child vertices. We denote by $Space(V)$ the set of all valid solutions represented by vertex $V$. Thus, for a parent vertex $V$, branching subdivides the solution space $Space(V)$ among a set of child vertices. In other words, the union of $\{Space(C) : C \text{ is a child of } V\}$ over all children of $V$ amounts to $Space(V)$. Bounding a vertex $V$ is the estimation of a value, $bound(V)$, that lower-bounds schedule lateness of all valid solutions in $Space(V)$. That is, $\forall \gamma \in Space(V) : \text{lateness}(\gamma) \geq \text{bound}(V)$. Bounding allows us to prune vertices whose bounds are higher (i.e., worse) than the lateness of the best solution found so far, say $BestLateness$, since such vertices cannot lead to an optimal solution. To enable pruning, a tentative schedule, $solution(V)$, is computed at each expanded vertex $V$ out of the set $Space(V)$. Vertices whose bound is greater than the lateness of $solution(V)$ are then pruned. The algorithm continues until an optimal solution is found, i.e., all vertices have been pruned except one, and no further branching is possible. The complete algorithm is thus listed below.

1. Set up $V_{root}$. Let $ActiveVertexSet = \{V_{root}\}$. Let $BestLateness = \text{lateness}(solution(V_{root}))$.
2. Let $V_{expand}$ be the vertex with the minimum $bound(V)$ among all vertices, $V \in ActiveVertexSet$. Pop $V_{expand}$ out of $ActiveVertexSet$.
3. Find $solution(V_{expand})$, and if $\text{lateness}(solution(V_{expand})) < BestLateness$, let $BestLateness = \text{lateness}(solution(V_{expand}))$. Find the children of vertex $V_{expand}$ applying the branching function, $branch(V)$ to $V_{expand}$.
4. Prune the vertices that do not improve on the best solution found so far, i.e., vertices $V$ for which $bound(V) \geq BestLateness$.
5. Add the remaining vertices to the set $ActiveVertexSet$.
6. If $ActiveVertexSet$ is not empty, go to step 2. Otherwise, return the solution with the current $BestLateness$.

Section 2.2.1 describes how the root vertex is set up. Section 2.2.2 describes the function $solution(V)$ that computes a schedule at vertex $V$ out of the space of valid schedules, $Space(V)$, represented by the vertex. Section 2.2.3 describes the branching function, $branch(V)$ that returns a set of child vertices given a parent vertex $V$. Section 2.2.4 describes the bounding function that determines, given some vertex $V$, a lower bound on schedule lateness for all schedules in $Space(V)$. These functions in conjunction with the pseudocode given above completely specify our B&B algorithm. As with any B&B algorithm, its optimality is guaranteed as long as 1) branching does not leave any part of the solution space unreachable, and 2) bounding computes a true lower bound of the performance measure for each vertex [13]. These properties are proven when discussing branching and bounding in Section 2.2.3 and Section 2.2.4, respectively.

**2.2.1 Setting up the Root Vertex**

The root vertex represents the entire space of all valid solutions. $Space(V_{root})$ is implicitly specified by 1) module arrival times and computation times, and 2) a set of synchronization constraints $Sync(V_{root}) = Sync_{init}$, as given in the problem input (e.g., see Table 2). Any valid solution to the scheduling problem must satisfy 1) and 2), (as well as account for message delay bounds computed by paradigm C1). A valid feasible solution would also satisfy all deadlines.

For the purpose of computing an initial schedule $solution(V_{root})$, we also compute an initial message priority order, $Ord(V_{root})$. In Section 2.2.2, we show how this information is used to obtain message delays and compute a valid schedule. $Ord(V_{root})$ can be altered during the search process to obtain better schedules as will be described in Section 2.2.3.

We compute the initial message priority order, $Ord(V_{root})$, using heuristic C3. The “urgency” of each message is first estimated as follows: The precedence constraints associated with messages among modules running on different PNs are neglected, and modules on each PN are scheduled using EDF subject to the remaining precedence and exclusion constraints. Let the resulting schedule be $\gamma_{no\ constraints}$. For each message $m_{i,j}$ from module $M_i$ to module $M_j$ we note the completion time $E_i(\gamma_{no\ constraints})$ of the sending module $M_i$ in schedule $\gamma_{no\ constraints}$, which is the time message $m_{i,j}$ is sent. The message must make it to the receiver $M_j$ in time for it to execute by its deadline, $D_j$. Thus, the relative deadline for message $m_{i,j}$ is set to $D_j - C_j - E_i(\gamma_{no\ constraints})$.

Messages are ordered by their relative deadlines, such that messages with tighter relative deadlines have higher priorities. Ties are broken arbitrarily. This results in the initial message priority order $Ord(V_{root})$. Fig. 1 demonstrates the schedule $\gamma_{no\ constraints}$ for one planning cycle of the task set in Table 2. It shows the intervals from message transmission time to receiver deadline for the two messages in the planning cycle, $m_{7,11}$ and $m_{9,4}$. From the figure it is seen that $m_{7,11}$ must have higher priority than $m_{9,4}$. Thus, $Ord(V_{root}) = m_{7,11}, m_{9,4}$ (in decreasing priority order).

**2.2.2 Computing solution(V)**

In order to prune vertices in the search space we compute a valid solution at each visited vertex $V$. The solution is drawn from the solution subspace $Space(V)$ represented by the vertex. The lateness of the computed solution is used to prune vertices whose $bound()$ is higher. Thus, the goodness of the function $solution(V)$ in picking a low-lateness schedule out of $Space(V)$ determines how efficient the pruning process is. In this subsection we describe the function $solution(V)$.

Given a complete message priority order $Ord(V)$ at vertex $V$, and a set $Sync(V)$ of synchronization constraints, we need to 1) compute message delays, 2) compute a valid task schedule, and 3) find schedule lateness. Of these, computing message delays is not performed by our algorithm. Instead, it is performed by the underlying communication paradigm, C1 (e.g., real-time channels). In our running example, we would thus use C1 to establish a
channel for the higher priority message \( m_{7,11} \) first, then establish a channel for message \( m_{9,4} \). For the purpose of illustration, assume that the delay bounds computed by \( C_1 \) for messages \( m_{7,11} \) and \( m_{9,4} \) are 1.75 and 3, respectively.

Once the message delay has been established for each message, the function \( \text{solution}(V) \) computes a new arrival time \( A_i \) and deadline \( D_i \) for each module \( M_i \) to account for message communication delays and precedence constraints. The following recursive equations are used.

\[
A_i = \max(A_i, \{ A_j + C_j + m_{ji} \mid M_j \text{ precedes } M_i \}) \tag{1}
\]

\[
D_i = \min(D_i, \{ D_j - C_j - m_{ij} \mid M_i \text{ precedes } M_j \}) \tag{2}
\]

where \( C_i \) is the worst-case computation time of \( M_i \), and \( m_{ij} \) (\( m_{ji} \)) is the computed communication delay between \( M_j \) and \( M_i \) (\( M_i \) and \( M_j \)), if any. This is a standard technique for deadline and release time modification in order to account for precedence constraints. Variations of it have been used in several publications, e.g., [5], [6], [10], [14]. For example, in the task set of Table 2,

\[
A_{11} = \max(0, A_7 + C_7 + m_{7,11}) = 2.75,
\]

and

\[
D_{11} = \min(12, D_{11} - C_{11} - m_{7,11}, D_8 - C_8) = \min(12, 9 - 1.75, 12 - 2) = 6.25.
\]

Next, \( \text{solution}(V) \) computes an EDF schedule subject to the above arrival times and deadlines, as well as precedence and exclusion constraints in set \( \text{Sync}(V) \). We choose EDF because it is a locally optimal preemptive scheduling policy. To prevent unbounded (dynamic) priority inversion, a module which blocks others with earlier deadlines inherits the earliest of these deadlines. We call this policy \( \text{EDF with Deadline Inheritance} \) (EDF-DI). Note that we do not use dynamic priority ceilings [15], because deadlocks cannot occur in our simplified model. Fig. 2 shows the schedule computed by \( \text{solution}(V) \) at the root vertex for the task set in Table 2. In this example, schedule lateness is \( \text{lateness}(\text{solution}(V_{root})) = 1.5 \), which is the lateness of \( T_5 \) (module \( M_{11} \)).

2.2.3 \textit{Branching} \( \text{branch}(V) \)

The branching function, \( \text{branch}(V) \), subdivides the valid solution space \( \text{Space}(V) \) represented by a parent vertex \( V \) into a set of subspaces, each represented by a child vertex. To make sure that no parts of the solution space are “lost,” the union of the subspaces \( \text{Space}(C) \) over all children \( C \) of vertex \( V \) must amount to \( \text{Space}(V) \). The set of valid solutions \( \text{Space}(V) \) at an arbitrary vertex \( V \) is implicitly represented by 1) module arrival times and computation times at vertex \( V \), and 2) the set of synchronization constraints \( \text{Sync}(V) \). The solution space is subdivided by the branching function in order to help pruning subsets of that space. We prune vertices whose lower bound is worse (i.e., higher) than the best lateness of a schedule found by \( \text{solution}(V) \). Thus, we need \( \text{solution}(V) \) to return progressively better schedules as we get deeper in the search tree to enable pruning more vertices. In what follows, we describe how branching is done, and present the methodology used to attempt to improve the lateness of \( \text{solution}(V) \) at each descendant.

Consider some vertex \( V \) in the search tree generated by our B&B algorithm. Let \( T_{m} \) be the task with the maximum task lateness in the schedule computed by \( \text{solution}(V) \). If there is a tie, let \( T_{m} \) be the task with the maximum lateness that finishes first. This tie-breaking rule is important to guarantee “progress” (i.e., prevent an infinite loop in the algorithm) as will be described later in this section. Let \( M_{last_m} \) be the last module of task \( T_{m} \). For example, in the root schedule shown in Fig. 2, \( M_{last_m} \) is \( M_{11} \). Unless the schedule at \( V \) happens to be optimal, there exists a way to reduce schedule lateness. In other words, it is possible to let module \( M_{last_m} \) finish earlier. In the following, we consider all possible ways of letting \( M_{last_m} \) finish earlier. In order to categorize and describe these ways, we use the concept of a \textit{busy period} [2]. Informally, the \textit{busy period}, \( B_i \), of module \( M_i \) is the interval \( [G_i, E_i] \), where \( E_i \) is the completion time of \( M_i \), and \( G_i \) is the start of the period of \textit{continuous} processor utilization that includes \( M_i \). For example, in the root schedule shown in Fig. 2, the busy period of the latest
module \( M_{11} \) is \( B_{11} = [6, 10.5] \). The busy period \( B_i \) of module \( M_i \) is defined recursively as follows:

1. \( M_i \in B_i \).
2. While \( \exists M_{i_k} \) whose completion time satisfies \( t < E_k < E_i \) (where \( t = \min \{ A_j \mid M_j \in B_i \} \) let \( M_{i_k} \in B_i \).

In order to reduce schedule lateness, we consider the following three cases, exactly one of which will be satisfied in any schedule (since their ORing amounts to unity).

Case 1: No module \( M_i \) in \( B_{\text{last}} \) has predecessors on other processors (according to set \( \text{Sync}(V) \)), and no module \( M_i \) in \( B_{\text{last}} \) has a deadline \( D_i > D_{\text{last}} \).

Case 2: \( \exists \) some module \( M_i \) in \( B_{\text{last}} \) whose deadline \( D_i > D_{\text{last}} \).

Case 3: \( \exists \) some module \( M_i \) in \( B_{\text{last}} \) who has a predecessor \( M_j \) on a different processor (according to set \( \text{Sync}(V) \)), and no module \( M_i \) in \( B_{\text{last}} \) has a deadline \( D_i > D_{\text{last}} \).

Case 1: No module \( M_i \) in \( B_{\text{last}} \) has predecessors on other processors (according to set \( \text{Sync}(V) \)), and no module \( M_i \) in \( B_{\text{last}} \) has a deadline \( D_i > D_{\text{last}} \).

Case 2: \( \exists \) some module \( M_i \) in \( B_{\text{last}} \) whose deadline \( D_i > D_{\text{last}} \).

Case 3: \( \exists \) some module \( M_i \) in \( B_{\text{last}} \) who has a predecessor \( M_j \) on a different processor (according to set \( \text{Sync}(V) \)), and no module \( M_i \) in \( B_{\text{last}} \) has a deadline \( D_i > D_{\text{last}} \).

Case 2: There exists some module \( M_i \) in \( B_{\text{last}} \) whose deadline \( D_i > D_{\text{last}} \). Since EDF scheduling is used to obtain \( \text{solution}(V) \), modules scheduled before the latest module \( M_{\text{last}} \) in its busy period must necessarily have earlier deadlines. The only way some module \( M_i \) in \( B_{\text{last}} \) can have \( D_i > D_{\text{last}} \) yet be scheduled before \( M_{\text{last}} \) is because of priority inversion due to a mutual exclusion constraint which prevents that module from being preempted. In other words, Case 2 implies that \( \exists M_j \in B_{\text{last}} \) such that \( M_i \text{ excludes } M_j \in \text{Sync}(V) \). For example, in Fig. 2 we can see a situation where the latest module \( M_{11} \) which becomes ready upon delivery of message \( m_{7,11} \) at time \( t = 6.75 \) cannot preempt a less urgent module \( M_{10} \) due to a mutual exclusion constraint, even though \( D_{10} > D_{11} \). We also see a case where \( M_{13} \) of intermediate priority delays less urgent \( M_{10} \) before \( M_{11} \) becomes ready, thus indirectly delaying the higher priority module \( M_{11} \) because of the mutual exclusion constraints. 4 Eliminating the exclusion constraint would resolve the problem of priority inversion, potentially resulting in a better schedule. To eliminate an exclusion constraint \( M_i \text{ excludes } M_j \), the branching function \( \text{branch}(V) \) generates two children \( C_1 \) and \( C_2 \) such that \( \text{Space}(C_1) \) is the subset of all schedules in \( \text{Space}(V) \) where \( M_i \text{ precedes } M_j \), and \( \text{Space}(C_2) \) is the subset of all schedules in \( \text{Space}(V) \) where \( M_i \text{ precedes } M_j \). In other words, we set the children’s synchronization constraints such that

\[
\text{Sync}(C_1) = \text{Sync}(V) - \{ M_i \text{ excludes } M_j \}
+ \{ M_i \text{ precedes } M_j \}
\]

and

\[
\text{Sync}(C_2) = \text{Sync}(V) - \{ M_i \text{ excludes } M_j \}
+ \{ M_i \text{ precedes } M_j \}
\]

Since an exclusion constraint \( M_i \text{ excludes } M_j \in \text{Sync}(V) \) means that in any valid solution, either \( M_i \text{ precedes } M_j \) or \( M_j \text{ precedes } M_i \), it can be seen that

\[
\text{Space}(C_1) \cup \text{Space}(C_2) = \text{Space}(V).
\]

However, in each child independently, the exclusion constraint has been replaced with a precedence constraint.

For the sake of illustration, Fig. 3 gives the \( \text{Sync}() \) sets of the two children of the root vertex whose schedule, shown in Fig. 2, satisfies Case 2. The figure shows that applying the \( \text{solution}() \) function to each child gives a better schedule (in terms of maximum task lateness) than that of the parent. (Compare the maximum lateness of the children in Fig. 3 to

4. EDF DIs does not prevent such priority inversion, because it cannot keep \( M_{10} \) from being preempted before it inherits the deadline of \( M_{11} \).
Child 1:

\[
\begin{array}{cccccccccccc}
T1 & T2 & T1 & T3 & T1 & T3 & T2 & T1 \\
M1 & M5 & M2 & M7 & M8 & M3 & M8 & M6 & M4 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12
\end{array}
\]

\[
T6 & T4 & T4 & T5 & T6 \\
M12 & M9 & M10 & M11 & M13 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12
\]

\[\text{Ord}(V) = m_{7,11}, m_{9,4}\]

\[\text{Sync}(V) = \{M_7 \text{ precedes } M_{11}, M_7 \text{ precedes } M_8, M_9 \text{ precedes } M_4, M_9 \text{ excludes } M_{11}, M_{10} \text{ precedes } M_{11}\}\]

\[\text{lateness}(\text{solution}(V)) = 1\]

Child 2:

\[
\begin{array}{cccccccccccc}
T1 & T2 & T1 & T3 & T1 & T3 & T2 & T1 \\
M1 & M5 & M2 & M7 & M8 & M3 & M8 & M6 & M4 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12
\end{array}
\]

\[
T6 & T4 & T5 & T6 & T4 \\
M12 & M9 & M11 & M13 & M10 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12
\]

\[\text{Ord}(V) = m_{7,11}, m_{9,4}\]

\[\text{Sync}(V) = \{M_7 \text{ precedes } M_{11}, M_7 \text{ precedes } M_8, M_9 \text{ precedes } M_4, M_9 \text{ excludes } M_{11}, M_{11} \text{ precedes } M_{10}\}\]

\[\text{lateness}(\text{solution}(V)) = 1.25\]

Fig. 3. Branching in Case 2.

that of root schedule in Fig. 2.) Child$_1$ results in decreasing the deadline of module $M_{10}$ (see (2)), thus preventing $M_{13}$ from preempting $M_{10}$. As a result, both $M_{10}$ and $M_{11}$ finish earlier, thus reducing schedule lateness. Child$_2$ results in causing $M_{10}$ to wait until $M_{11}$ is finished instead of blocking it due to mutual exclusion, also reducing schedule lateness. This lateness improvement leads to more efficient pruning. Intuitively, the improvement is attributed to the local optimality of EDF scheduling when no mutual exclusion constraints are present.

Case 3: There exists some module $M_i$ in $B_{last_m}$ who has a predecessor $M_j$ on a different processor (according to set $\text{Sync}(V)$), and no module in $B_{last_m}$ has a deadline $D_i > D_{last_m}$. Following the same argument as in Case 1, the schedule generated in Case 3 for the busy period $B_{last_m} = [G_{last_m}, E_{last_m}]$ is locally optimal. The latest module has the largest deadline and is therefore scheduled last in the busy period. However, since in Case 3, $\exists M_i \in B_{last_m}$ which has a remote predecessor $M_j$, schedule lateness may be improved by either increasing the priority of message $m_{j,i}$ if any, to let it arrive earlier at the destination, or else by rescheduling $M_i$, earlier on its processor. In either case, the busy period $B_{last_m}$ as a whole may start earlier, thus potentially reducing schedule lateness. For example, consider the schedule of Child$_2$ shown in Fig. 3. The latest module $M_{last_m}$ is $M_{10}$ whose lateness is 1.25. The busy period $B_{last_m}$ is $B_{10} = [6.75, 11.25]$. There exists a module $M_i = M_{11}$ in $B_{last_m}$ with a remote predecessor $M_j = M_7$, thus Case 3 is satisfied. Message $m_{7,11}$ already has top priority. So we consider scheduling $M_7$ earlier to decrease the lateness of the latest task. In general, a remote predecessor $M_j$ is forced to start earlier by decreasing its deadline such that it inherits the lateness of $M_{last_m}$. Thus, the branching function in Case 3 returns a set of children as follows:
If a message \( m_{i;j} \) from a remote predecessor \( M_j \) immediately preceding some \( M_i \) in \( B_{\text{last}_m} \) (i.e., \( M_j \) precedes \( M_i \) in \( \text{Sync}(V) \)), increase the priority of \( m_{i;j} \) (if possible). The algorithm shown in Fig. 4 specifies how message priority is increased. Essentially, the first promoted message gets the highest priority. Each time another message is promoted, its priority is set one level below the priority of the previously promoted message. The variable \( \text{priority}_{\text{limit}}(V) \) tells which level message priority should be increased to at search vertex \( V \). The variable is set to the highest priority, \( 1 \), at the root, and is incremented each time a message has been promoted. We do not claim optimality with respect to setting message priorities, although we expect that increasing the priority of a message on the critical path is likely to improve schedule lateness.

Fig. 4. The branching function.

- If \( \exists \) a message \( m_{i;j} \) from a remote predecessor \( M_j \) immediately preceding some \( M_i \in B_{\text{last}_m} \) (i.e., \( M_j \) precedes \( M_i \in \text{Sync}(V) \)), increase the priority of \( m_{i;j} \) (if possible). The algorithm shown in Fig. 4 specifies how message priority is increased. Essentially, the first promoted message gets the highest priority. Each time another message is promoted, its priority is set one level below the priority of the previously promoted message. The variable \( \text{priority}_{\text{limit}}(V) \) tells which level message priority should be increased to at search vertex \( V \). The variable is set to the highest priority, \( 1 \), at the root, and is incremented each time a message has been promoted. We do not claim optimality with respect to setting message priorities, although we expect that increasing the priority of a message on the critical path is likely to improve schedule lateness.

- If message priority cannot be increased (or there are no messages), then for all remote predecessors \( M_j \) immediately preceding some \( M_i \in B_{\text{last}_m} \) (i.e., \( M_j \) precedes \( M_i \in \text{Sync}(V) \)) create a child vertex with same module arrival times, computation times, and synchronization constraints, and let \( M_j \) inherit the lateness of the latest task, i.e., reduce the deadline of \( M_j \) in the child to \( \min(D_j, E_j - \text{lateness}(M_{\text{last}_m})) \), where \( \text{lateness}(M_{\text{last}_m}) = E_{\text{last}_m} - D_{\text{last}_m} \). Note that changing a module deadline makes the heuristic function, \( \text{solution}() \), return a different (potentially better) EDF schedule for child \( C \) than it does for its parent \( V \), leading to pruning more vertices. However, \( \text{Space}(C) = \text{Space}(V) \) because the valid solution space is defined independently of deadline values.

If the deadline of a predecessor \( M_j \) of the latest module is advanced, the predecessor will either:

1. Finish earlier in the new schedule (i.e., a different schedule, \( \gamma_C \), results at the child), or
2. Become the latest task itself.

In either case “progress” is guaranteed in the sense that the child differs from the parent in either the schedule returned by \( \text{Solution}(V) \) or the latest task. Note that option 2 above is because the predecessor’s deadline has been advanced to \( D_j = E_j(\gamma_V) - \text{lateness}(M_{\text{last}_m}) \), where \( \gamma_V \) is the schedule at the parent vertex \( V \). Thus, if the predecessor does not finish earlier in the new schedule (i.e., if \( E_j(\gamma_C) = E_j(\gamma_V) \)), its lateness will be

\[ E_j(\gamma_C) - D_j = E_j(\gamma_V) - D_j = \text{lateness}(M_{\text{last}_m}). \]

Since our tie breaking rule chooses the latest task to be the one with the minimum completion time among those with the maximum lateness, it will choose \( M_j \) rather than \( M_{\text{last}_m} \) as the latest task in the new schedule, \( \gamma_C \). This guarantees progress. Note that the branching in Case 3 is similar to substituting a precedence constraint between the latest task and one of its remote predecessors with an artificial deadline on the predecessor. Removal of a precedence constraint between tasks residing on different processors brings the EDF schedule at the child vertex closer to the optimal. EDF is optimal when all such precedence constraints (and all exclusion constraints) are removed. Fig. 5 illustrates the branching in Case 3. It shows the schedule obtained by branching from vertex Child2 shown in Fig. 3 by letting the predecessor \( M_7 \) inherit the lateness of the latest module \( M_{\text{last}_m} \). The new deadline of \( M_7 \) becomes \( D_7 = 5 - 1.25 = 3.75 \). Intuitively,

![Fig. 5. Branching in Case 3.](image-url)
must complete by that deadline in order for M to complete in time. The resulting new EDF schedule shifts M earlier (in accordance with its new deadline), which happens to result in a globally optimal schedule. Fig. 6 summarizes the branching function.

2.2.4 Bounding bound(V)

Our bounding function determines a lower bound on lateness at a vertex V by removing from set Sync(V) 1) all the mutual exclusion constraints, and 2) all precedence constraints among modules on different processors, then computing vertex cost subject to the remaining set of local precedence constraints, say Syncren(V) \subset Sync(V), and module arrival times and deadlines as described in Section 2.2.2. (Note that message priorities are irrelevant here, because precedence constraints and delays associated with interprocessor messages have been ignored in Syncren(V).)

The lateness of the computed EDF schedule is globally optimal among all schedules that satisfy Syncren(V). This is because:

1. Since no exclusion constraints are present in Syncren(V), EDF is locally optimal.
2. Since all constraints in Syncren(V) are between modules on the same processor, modules on each processor are “independent” of modules on every other processor.

Therefore, the set of locally optimal uniprocessor schedules is a globally optimal schedule. Let the aforementioned optimal lateness be called Lmin.

Finally, since any solution S ∈ Space(V) to the original task scheduling problem satisfies the constraint set Sync(V), it satisfies, by implication, the constraint subset Syncren(V). Thus, the lateness of S cannot be less than the global optimum Lmin. Thus, Lmin is a true lower-bound on lateness for any valid solution in Space(V). The optimality of the B&B algorithm follows from the correctness of the bounding function, and the ability of the branching function to encompass the entire solution space. Intuitively, the branching function transforms the set of initial precedence and exclusion constraints into an equivalent set of constraints with no mutual exclusion and no precedence constraints across different processors; a case in which EDF is globally optimal.

3 Evaluation

To demonstrate the utility of the algorithm, a simulator was constructed, generating arbitrary task graphs on which the algorithm can be applied. On each run, the algorithm was given a task graph, an optimal solution was found, and the number of generated vertices was recorded. The numbers of modules, processors, precedence and exclusion constraints, were varied to determine the trends in algorithm performance. We restrict the following discussion to the most tangible results we found, namely, the effects of application concurrency, CPU utilization, and module interaction on the performance of the algorithm. We considered systems of 300 modules running on four processors. The effect of application concurrency is analyzed by varying the number of concurrent application threads per processor. CPU utilization is varied by changing the average module execution time. Finally, module interaction is controlled by varying the number of communicated messages between different modules. In the figures given below, each point is obtained by averaging 25 readings.

Fig. 7 shows the effect of concurrency. The number of concurrent application threads per processor is varied from one to nine. Note that the total number of application threads may be much more. By concurrent threads we mean only those that become ready to execute during the same time intervals. CPU utilization is fixed at 90 percent, and the number of messages in the system is fixed at 150 (half the number of modules). It can be seen that an optimal schedule is found near the root in most cases when the degree of concurrency is low. This is explained by the fact that EDF scheduling (performed at the root) is locally optimal.
Exploiting this characteristic may lead to an optimal solution when application concurrency is low. For concurrency of six threads per processor or less, the average number of generated vertices is less than five. As the concurrency increases, the local optimality of EDF scheduling becomes less and less sufficient. Thus, our algorithm expands progressively more vertices to find a globally optimal solution.

Fig. 8 demonstrates the effect of CPU utilization. CPU utilization is the total computational workload per processor divided by schedule length. The number of messages in the system was fixed at 150, and the average degree of concurrency was eight. The algorithm performs very well for utilization up to 80 percent. The average number of generated vertices over that range is less than 10. As utilization increases, the algorithm runtime increases abruptly, due mainly to the accompanying increase in the length of the busy period, and therefore the increase in branching factor.

Finally, Fig. 9 illustrates the effect of module interaction measured in the number of communicated messages within the system. CPU utilization was fixed at 90 percent, and the degree of concurrency was fixed at eight. Unlike the other curves, the algorithm runtime increases almost linearly with the number of messages, because, as the number of messages increases, so does the branching factor. However, since each message introduces a precedence constraint, increasing the number of messages tends to constrain the task graph and decrease scheduling options at any given time, thus reducing the depth of the search tree.

In general, for a wide range of workloads the algorithm generates an optimal solution at or near the root of the search tree. A similar observation was reported in [3]. This is due to the nature of the performance measure being optimized. Schedule lateness refers to the lateness of only one module. If we happen to be unable to reduce the lateness of the latest module, the algorithm terminates even though there may be ways of decreasing the lateness of other modules. The generation of vertices is inexpensive. For the task sets considered in this section, the worst-case computation time of a run was in the order of a few seconds on a Sun Ultra workstation.

4 A SIMPLE HEURISTIC

The complexity of the algorithm presented in Section 2 can be reduced by employing a greedy heuristic which performs depth-first search with no backtracking. Our heuristic expands each vertex $V$ by generating all its children, then branches to the minimum-cost child, ignoring all others. Thus, at each vertex, after generating its children,
a complete schedule is computed for each child (as opposed to a lower bound). Children whose schedule lateness is more than that of the parent are pruned. Among the surviving ones, the child with the minimum schedule lateness is selected for expansion next. The algorithm continues until a vertex is reached that has no children (or until the first feasible schedule is found, if so desired).

Although pathological cases may be constructed where the heuristic fails to find an existing feasible schedule, it was able to arrive at the optimal schedule in 29 out of 30 randomly generated cases of periodic task sets with 90 percent CPU utilization. This number, however, depends much on the nature of the task set. In the case where tasks arrive at random times with random deadlines, the heuristic performs worse than in the case where tasks arrive at regular intervals and have the same deadline at each invocation. To compare the costs of running the two algorithms, 30 randomly generated periodic task sets (of 400 tasks each) were constructed, and each algorithm was run on each set. The number of generated vertices was not used as a measure for algorithm comparison, since the amount of computation per vertex is higher for the heuristic algorithm.

This is because it computes a complete schedule for each child vertex, while the optimal one computes only a lower bound. Since the computation of a schedule at a vertex was found to be the most costly element of both B&B algorithms, the number of complete schedules computed until a solution is found was taken as the measure for their comparison. The heuristic was found to generate 74 percent fewer schedules than the optimal algorithm for the task set size considered before the best schedule is found. It is projected that the savings are greater for larger task sets.

5 A MORE GENERAL RESOURCE CONSTRAINT MODEL

In the previous sections, we presented an algorithm for combined task and message scheduling in distributed real-time systems. The algorithm uses a simple model for resource requirements. The model has two limitations, i.e., resources are assumed to be:

- locked/unlocked together;
- all locked at module start and unlocked upon module termination.

Fig. 10 illustrates a consequence of these limitations. For example, resource R2 has to be locked by the modules throughout the entire interval of their execution, even though it is used only for a part of that interval. A more severe consequence is that all resources have to be unlocked at module termination, thus in Fig. 10 the exclusion constraints do not prevent the scheduler from inserting a module which uses R1 or R2 between M₁ and M₂, which should not be the case. In what follows, we present a more realistic version of resource constraints, and describe how the algorithm in Section 2 can be modified to accommodate it.

5.1 A General Exclusion Model

We extend the notion of an exclusion constraint to include exclusion between strings of modules where a string of k modules $M₁ M₂ \cdots Mₖ$ is a module sequence in which $M₁ \text{ precedes } M₂ \cdots \text{ precedes } Mₖ$. Thus, the constraints have the general form

[Diagram of a more general exclusion model]
meaning that all modules in one of the strings have to terminate before any module in the other can start. For example, Fig. 11 depicts the resource requirements of two tasks. The following exclusion constraints can be derived:

- **C1**: \( M_2 M_3 M_4 M_5 \) excludes \( M_6 M_{10} \) (because of R1)
- **C2**: \( M_2 M_4 \) excludes \( M_6 M_9 M_{10} M_{11} \) (because of R2)
- **C3**: \( M_5 \) excludes \( M_8 M_9 M_{10} M_{11} \) (because of R3)

Note that under this model, a deadlock may occur. For example, in Fig. 11 a potential deadlock arises because task \( \text{task}_1 \) and task \( \text{task}_2 \) lock resources R1 and R2 in different orders.

### 5.2 Algorithm Modifications

A close look at the algorithm may reveal that no modifications are necessary to accommodate the new type of exclusion constraints. This is because an exclusion constraint such as \( M_2 M_3 M_4 M_5 \) excludes \( M_6 M_{10} \) can simply be viewed as a simple exclusion constraint \( M_i \) excludes \( M_j \), where \( M_i = M_2 M_3 M_4 M_5 \), and \( M_j = M_6 M_{10} \). The algorithm can account for such constraints as discussed earlier. Our only concern is to avoid deadlocks when computing a valid solution, \( \text{Solution}(V) \). Thus, to accommodate the exclusion model presented above, we slightly modified the definition of a valid schedule. In particular, in addition to the former requirements, a valid schedule must also be deadlock-free. As a result, the function \( \text{Solution}(V) \) must have a way to detect deadlocks in the computed solution. This function is performed off-line, and can utilize any of the known methods for deadlock detection. (Deadlock detection is not a concern of this paper.) The problem is how to modify the schedule to avoid the deadlock without losing optimality.

Note that if there were no exclusion constraints, deadlocks would not develop. We had already demonstrated one way of removing exclusion constraints by replacing them with precedence constraints. When a deadlock is detected by \( \text{Solution}(V) \), it is circumvented using a similar technique. Consider a deadlock that occurs during the computation of a solution schedule \( \text{Solution}(V) \) at some search vertex \( V \). A typical deadlock detection algorithm can then identify a cycle of modules in which each module cannot run because it is waiting for another module in the cycle to proceed. The presence of the cycle manifests the deadlock. Since there are no cycles in the precedence constraint graph, one of the edges in the deadlock cycle must be due to an exclusion constraint between the corresponding modules. This exclusion constraint can be replaced by one of two possible precedence constraints,\(^5\) thus splitting the current vertex into two. If the exclusion constraint replacement results in a cyclic precedence constraint graph in any of the two generated vertices, this vertex is destroyed, since it does not lead to valid solutions. Bounding is then performed on the surviving of the two vertices, and the one with the lower bound is expanded. \( \text{Solution}(V) \) is then applied to the expanded vertex. It will no longer run into the same deadlock. In essence, to avoid the deadlock we have “eliminated” an exclusion constraint the same way we described earlier in the context of branching, albeit for a different reason. Eventually enough of the exclusion constraints will be eliminated to produce a deadlock-free schedule.

### 6 Conclusion

This paper presented a new B&B algorithm for off-line combined task and message scheduling in distributed systems. The algorithm computes a schedule for tasks, a set of deadlines for messages derived from a priority assignment, and a route selection such that the maximum task lateness is minimized. It accounts for precedence and exclusion constraints between task modules. Furthermore, it exploits the coupling between task completion times and message delays by recomputing message priorities and deadlines during the search to reduce the maximum task lateness.

The algorithm is proven to compute an optimal schedule for the set of message deadlines defined for the solution point in the search space. A simulation study has shown that it scales well with respect to system size and the number of communicated messages among tasks. A heuristic version of the algorithm is presented, where a greedy technique is used to trade optimality for speed. We also suggest a way to generalize the algorithm for a more practical resource mode.

### ACKNOWLEDGMENTS

An earlier version of this paper was presented at the 1995 IEEE Real-Time Systems Symposium. The work reported in this paper was supported in part by the U.S. Office of Naval Research under Grant N00014-99-1-0465, and the U.S. National Science Foundation under grant MIP-9203895. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of funding agencies.

### References


---

\(^5\) The replacement of an exclusion constraint by complementary precedence constraints has been described in Section 2.2.3.


Tarek Abdelzaher received his BSc and MSc degrees in electrical and computer engineering from Ain Shams University, Cairo, Egypt, in 1990 and 1994, respectively. In 1994, he began his PhD studies with the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor, and received his PhD in 1999. He is currently an assistant professor for the Department of Computer Science at the University of Virginia. His research interests are in the field of Quality of Service (QoS) provisioning and real-time computing. He is a member of the IEEE Computer Society and a recipient of the Distinguished Student Achievement Award in Computer Science and Engineering.

Kang G. Shin received the BS degree in electronics engineering from Seoul National University, Seoul, Korea in 1970, and the MS and PhD degrees in electrical engineering from Cornell University, Ithaca, New York, in 1976 and 1978, respectively. From 1978 to 1982, he was on the faculty of Rensselaer Polytechnic Institute, Troy, New York. He has authored/coauthored more than 600 technical papers and numerous book chapters in the areas of distributed real-time computing and control, computer networking, fault-tolerant computing, and intelligent manufacturing. In 1985, he founded the Real-Time Computing Laboratory, where he and his colleagues are investigating various issues related to real-time and fault-tolerant computing. He received the Outstanding IEEE Transactions on Automatic Control Paper Award in 1987, and the Research Excellence Award from The University of Michigan in 1989. He is currently at the University of Michigan, Ann Arbor, where his research focuses on QoS sensitive computing and networking, with emphasis on timeliness and dependability. He has also been applying the basic research results to telecommunication and multimedia systems, intelligent transportation systems, embedded systems, and manufacturing applications. Dr. Shin is a fellow of the IEEE.