Optimal Combined Task and Message Scheduling in Distributed Real-Time Systems

Tarek F. Abdelzaher and Kang G. Shin

Real-time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109–2122
{*zaher,kgshin*}@*eecs.umich.edu*
734-763-0391 (voice); 734-963-4617 (fax)

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 an *optimal* 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 has the property of generating a complete schedule at each vertex in the search tree, and can be made to yield a feasible solution (found before reaching an optimal solution), or proceed until an optimal solution is found.

We have conducted an extensive simulation study evaluating 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 does not guarantee optimality. We have also extended the algorithm to a more general resource-constraint model thus widening its application domain.

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

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 Advanced Research Project Agency unde grant F30602-95-1-0044 and the National Science Foundation under grant MIP–9203895. Any opinions, findngs, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of funding agencies.

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 pre-runtime 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 accounted for. Furthermore, the algorithm should find a feasible schedule (i.e., one where all deadlines are met), 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 non-positive. If the optimal schedule has positive lateness, then we know that no feasible schedule exists. Thus, the problem is to find the optimal[1] pre-runtime 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 inter-machine messages.

In distributed hard real-time systems, inter-machine message communication affects task schedulability, and thus, has to be accounted for. 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, and task

[1] in the sense of minimizing maximum task lateness

lateness in the corresponding task schedules,[2] the problems of task scheduling and message scheduling 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 pseudo deadlines of tasks and messages, then employs an on-line guarantee routine to find a run-time 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 reduces 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, indeed, there are no feasible solutions to the problem or not.

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 it is presented in Section 4. It uses a greedy (instead of optimal) technique 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] Actually, there exists a circular dependency.

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, P, of p *processing nodes* (PNs), $PN_1, \cdots, PN_p$, connected by an arbitrary network N. PNs run a set T of n hard real-time tasks, $T_1, \cdots, T_n$. 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 T_k in the distributed system has a *known* arrival time a_k, total execution time c_k, and deadline d_k. In the case of periodic tasks, each task also has a period P_k. The arrival time of periodic task T_k is associated with its individual invocations, such that the arrival time $a_k[j]$ of its jth invocation, $T_k[j]$, is the beginning of its period $a_k = (j-1)P_k$. 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 such 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.

$Task_k$	PN	c_k	$d_k - a_k$	P_k
T_1	1	1	3	3
T_2	1	2	5.5	6
T_3	1	3	11	12
T_4	2	3	4	6
T_5	2	1	9	12
T_6	2	0.5	3.5	6

Table 1: An example task set.

A task may be composed of one or more *modules*. Each module M_i of a task invocation $T_k[j]$ has the worst-case execution time C_i which bounds its actual execution time, the arrival time A_i which denotes the earliest time the module can be invoked, and the deadline D_i which is the latest time it can finish execution. Initially, $A_i = a_k[j]$, and $D_i = d_k[j]$ of the corresponding task invocation. In a particular schedule γ, the time $S_i(\gamma)$ when module M_i is first given the CPU is called the *module start time* and the time $E_i(\gamma)$ when the module finishes execution is called the *module completion time*. The completion time of a task invocation $T_k[j]$ is the completion time E_{last} of its last module M_{last}. The *lateness* of task invocation $T_k[j]$ in schedule γ is defined as the lateness of the task's last module, $E_{last}(\gamma) - D_{last}$. A positive task lateness indicates that the task missed its deadline. Schedule lateness, $lateness(\gamma)$ 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 resource). A precedence constraint M_i *precedes* M_j means 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 $Sync_{init}$ is the set of all synchronization constraints defined initially for the system. In this section we cast resource constraints as mutual exclusion constraints between (entire) modules which implies that modules lock/unlock *all* their required resources *together*, and hold them throughout the entire interval of their execution. Thus, there is no possibility of deadlock. This restriction will be relaxed in Section 5. Modules residing on different processors communicate via message passing. A message from module M_i to module M_j in a planning cycle is denoted by $m_{i,j}$. Communication among modules residing on the same node is assumed to incur a fixed overhead, which is included in the execution time of the sending module.

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_5 (module M_{11}) use some results computed in module M_7 of task T_3 that are sent to T_5 via a message $m_{7,11}$. We also let each odd-numbered invocation of task T_4 (i.e., module M_9) communicate a message to the fourth invocation of task T_1 (module M_4) in the same planning cycle. Furthermore, we let tasks T_4 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 $Sync_{init}$ is shown in Table 2. This example task set will be used throughout the paper to illustrate the algorithm.

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

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

$Sync_{init} = \{M_7\ precedes\ M_{11}, M_7\ precedes\ M_8, M_9\ precedes\ M_4, M_9\ excludes\ M_{11}, M_{10}\ excludes\ M_{11}\}$

Table 2: The module set.

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 rather a parameter of the underlying system. It must guarantee bounded communication delay for messages. We compute (i) a message priority order using some heuristic C3 that attempts to reduce task lateness, (ii) message delay bounds using paradigm C1, and (iii) an optimal schedule using the optimal task scheduling algorithm C2 that minimizes task lateness 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.

A B&B technique is used to implement the above-mentioned approach. It can be viewed as a search, by implicit enumeration, through the entire *valid solution* space. A valid solution, γ, to the combined scheduling problem is a schedule with the following properties.

- Every module M_i in γ 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 $Sync_{init}$ are satisfied.[3]
- Messages 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 $S_j(\gamma) \geq d_{i,j} + E_i(\gamma)$.

A valid solution γ is *feasible* if it satisfies the additional constraint that every module M_i in γ 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. 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, $\forall\gamma \in Space(V) : lateness(\gamma) \geq bound(V)$. 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 $BestLateness = lateness(solution(V_{root}))$
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]We have defined in Section 2.1 what it means to satisfy the precedence and exclusion constraints.

3. Find $solution(V_{expand})$, and if $lateness(solution(V_{expand})) < BestLateness$ let $BestLateness = lateness(solution(V_{expand}))$. 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 non-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 pseudo-code given above completely specify our B&B algorithm. As with any B&B algorithm, its optimality is guaranteed as long as (i) branching does not leave any part of the solution space unreachable, and (ii) bounding computes a true lower bound of the performance measure for each vertex [13]. These properties are proved 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 (i) module arrival times and computation times, and (ii) 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 (i) and (ii), (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 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 $Ord(V_{root})$. Figure 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).

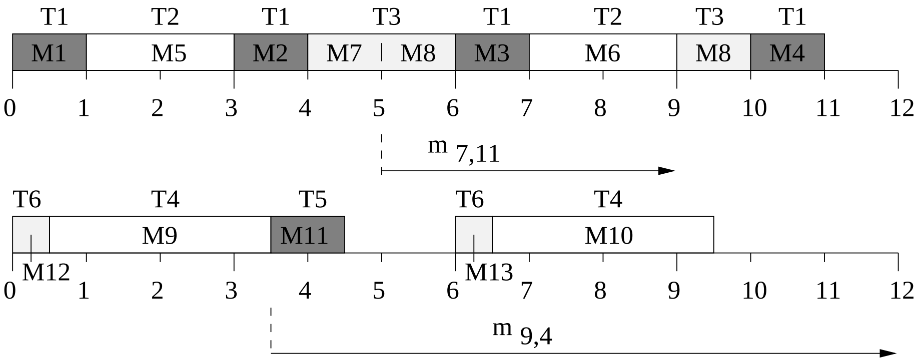


Figure 1: Computing message priority order $Ord(V_{root})$.

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 (i) compute message delays, (ii) compute a valid task schedule, and (iii) 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 C1 for messages $m_{7,11}$ and $m_{9,4}$ are 1.75 and 3, respectively.

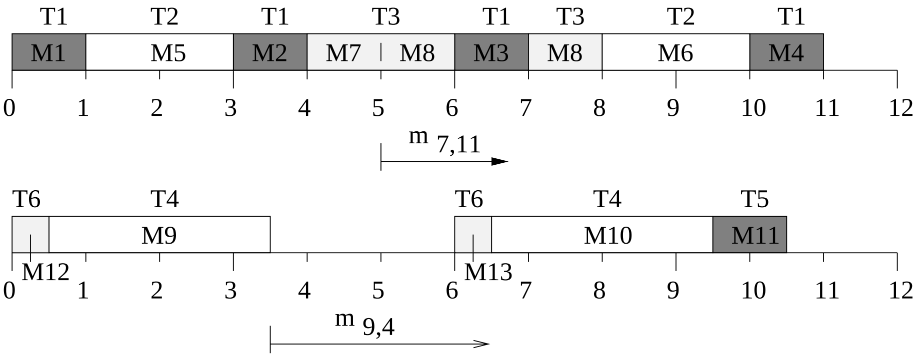
Once the message delay has been established for each message, the function $solution(V)$ computes a new arrival time A_i, and deadline D_i for each module M_i to account for message communication delays and precedence constraints. The following recursive equations are used.

$$A_i = \max(A_i,\ \{A_j + C_j + m_{ji} \mid M_j \ precedes \ M_i\}) \tag{2.1}$$

$$D_i = \min(D_i,\ \{D_j - C_j - m_{ij} \mid M_i \ precedes \ M_j\}). \tag{2.2}$$

where C_j is the worst-case computation time of M_j, and m_{ji} (m_{ij}) is the computed communication delay between M_j and M_i (M_i and M_j), if any. For example, in the task set of Table 2, $A_{11} = \max(0, A_7 + C_7 + m_{7,11}) = 2.75$, and $D_7 = \min(12, D_{11} - C_{11} - m_{7,11}, D_8 - C_8) = \min(12, 9 - 1 - 1.75, 12 - 2) = 6.25$

Next, $solution(V)$ computes an EDF schedule subject to the above arrival times and deadlines, as well as precedence and exclusion constraints in set $Sync(V)$. 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 *EDF with Deadline Inheritance* (EDF-DI). Note that we do not use dynamic priority ceilings [14], because deadlocks cannot occur in our simplified model. Figure 2 shows the schedule computed by $solution(V)$ at the root vertex for the task set in Table 2. In this example, schedule lateness is, $lateness(solution(V_{root})) = 1.5$, which is the lateness of T_5 (module M_{11}).



$Ord(V) = m_{7,11}, m_{9,4}$

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

$lateness(solution(V)) = 1.5$

Figure 2: Root schedule.

2.2.3 Branching $branch(V)$

The branching function, $branch(V)$, subdivides the valid solution space $Space(V)$ 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 $Space(C)$ over all children C of vertex V must amount to $Space(V)$. The set of valid solutions $Space(V)$ at an arbitrary vertex V is implicitly represented by (i) module arrival times and computation times at vertex V, and (ii) the set of synchronization constraints $Sync(V)$. 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 $solution(V)$. Thus, we need $solution(V)$ 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 $solution(V)$ 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 $solution(V)$. 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 Figure 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 find it useful to define the concept of a *busy period*. Let the *busy period*, B_i, of module M_i be 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 *continuous* processor utilization that includes M_i. For example, in the root schedule shown in Figure 2 the busy period of the latest module M_{11} is $B_{11} = [6, 10.5]$. Formally, the busy period B_i of module M_i is defined recursively as follows:

1. $M_i \in B_i$,
2. While $\exists\ M_k$ whose completion time satisfies $t < E_k < E_i$ (where $t = \min \{ A_j \ \ | \ M_j \in B_i) \}$) let $M_k \in B_i$.

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_{last_m} has predecessors on other processors (according to set $Sync(V)$), and no module M_i in B_{last_m} has a deadline $D_i > D_{last_m}$.

Case 2: $\exists$ some module M_i in B_{last_m} whose deadline $D_i > D_{last_m}$.

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

Case 1: No module M_i in B_{last_m} has predecessors on other processors (according to set $Sync(V)$), and no module M_i in B_{last_m} has a deadline $D_i > D_{last_m}$. In any schedule with a lower lateness than $solution(V)$ the module M_{last_m} (which has the maximum lateness in $solution(V)$) must complete earlier. However, since all other modules in B_{last_m} have tighter deadlines, the schedule where M_{last_m} executes last in B_{last_m} (i.e., $solution(V)$) is optimal. That is to say, it is the optimal solution within the subset $Space(V)$ of all possible schedules represented by vertex V. No further branching from that vertex is possible. The branching function returns a null set of children.

Case 2: There exists some module M_i in B_{last_m} whose deadline $D_i > D_{last_m}$. Since EDF scheduling is used to obtain $solution(V)$, modules scheduled before the latest module M_{last_m} in its busy period must necessarily have *earlier* deadlines. The only way some module M_i in B_{last_m} can have $D_i > D_{last_m}$ yet be scheduled before M_{last_m} 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 $\exists M_j \in B_{last_m}$ such that $M_i\ excludes\ M_j \in Sync(V)$. For example, in Figure 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, $M_i\ excludes\ M_j$, the branching function $branch(V)$ generates two children C_1 and C_2 such that $Space(C_1)$ is the subset of all schedules in $Space(V)$ where $M_i\ precedes\ M_j$, and $Space(C_2)$ is the subset of all schedules in $Space(V)$ where $M_j\ precedes\ M_i$. In other words, we set the children's synchronization constraints such that $Sync(C_1) = Sync(V) - \{M_i\ excludes\ M_j\} + \{M_i\ precedes\ M_j\}$ and $Sync(C_2) = Sync(V) - \{M_i\ excludes\ M_j\} + \{M_j\ precedes\ M_i\}$. Since an exclusion constraint $M_i\ excludes\ M_j \in Sync(V)$ means that in any valid solution either $M_i\ precedes\ M_j$ or $M_j\ precedes\ M_i$, it can be seen that $Space(C_1) \cup Space(C_2) = Space(V)$. However, in each child independently, the exclusion constraint has been replaced with a precedence constraint.

[4] EDF-DI does not prevent such priority inversion because it cannot keep M_{10} from being preempted *before* it inherits the deadline of M_{11}

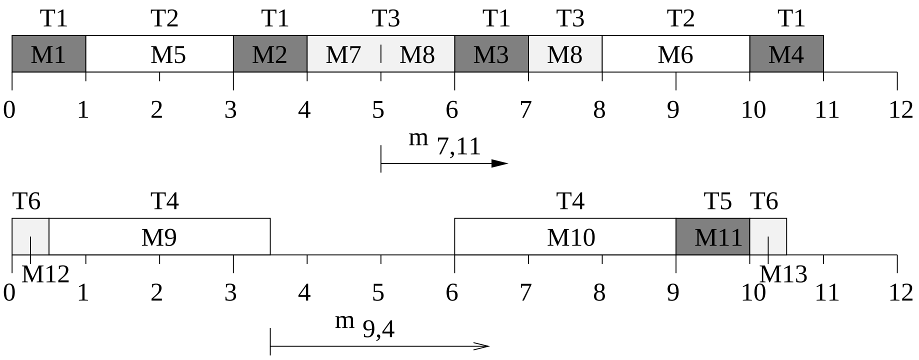
For the sake of illustration, Figure 3 gives the $Sync()$ sets of the two children of the root vertex whose schedule, shown in Figure 2, satisfies Case 2. The figure shows that applying the $solution()$ function to each child gives a better schedule (in terms of maximum task lateness) than that of the parent. (Compare the maximum lateness of the children in Figure 3 to that of root schedule in Figure 2.) $Child_1$ results in decreasing the deadline of module M_{10} (see Equation 2.2) 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 $Sync(V)$), 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_j 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 Figure 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 $\exists$ a message $m_{i,j}$ from a remote predecessor M_j immediately preceding some $M_i \in B_{last_m}$ (i.e., M_j *precedes* $M_i \in Sync(V)$) increase the priority of $m_{i,j}$ (if possible). The algorithm shown in Figure 4 specifies how message priority is increased. 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 $M_i \in B_{last_m}$ (i.e., M_j *precedes* $M_i \in Sync(V)$) 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

Child 1:

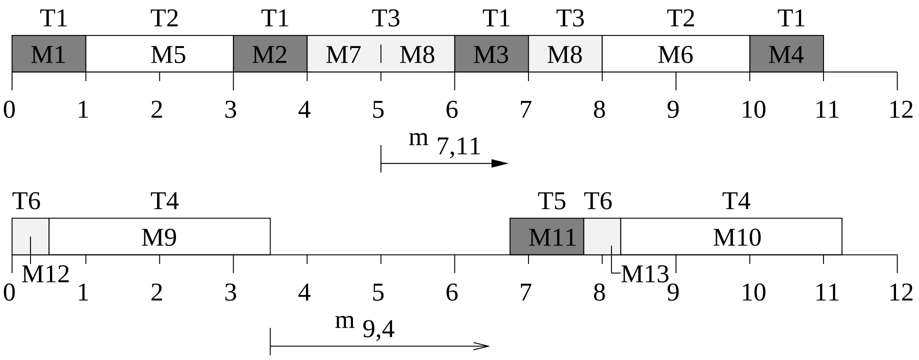


$Ord(V) = m_{7,11}, m_{9,4}$

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

$lateness(solution(V)) = 1$

Child 2:



$Ord(V) = m_{7,11}, m_{9,4}$

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

$lateness(solution(V)) = 1.25$

Figure 3: Branching in Case 2

$\min(D_j, E_j - lateness(M_{last_m}))$, where $lateness(M_{last_m}) = E_{last_m} - D_{last_m}$. Note that changing a module deadline makes the heuristic function, $solution()$, return a different (potentially better) EDF schedule for child C than it does for its parent V, leading to pruning more vertices. However, $Space(C) = Space(V)$ 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 (i) finish earlier in the new schedule (i.e., a different schedule, γ_C, results at the child), or (ii) will 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 $Solution(V)$ or the latest task. Note that case-(ii) above is because the predecessor's deadline has been advanced to $D_j = E_j(\gamma_V) - lateness(M_{last_m})$, where γ_V is

```
message_priority_increase(V)
 if V = V_root
   priority_limit (V) = 1
   let message_priority = priority_limit (V)
 else
   priority_limit (V) = priority_limit (parent (V)) + 1
   if message_priority is lower than priority_limit (V)
     let message_priority = priority_limit (V)
```

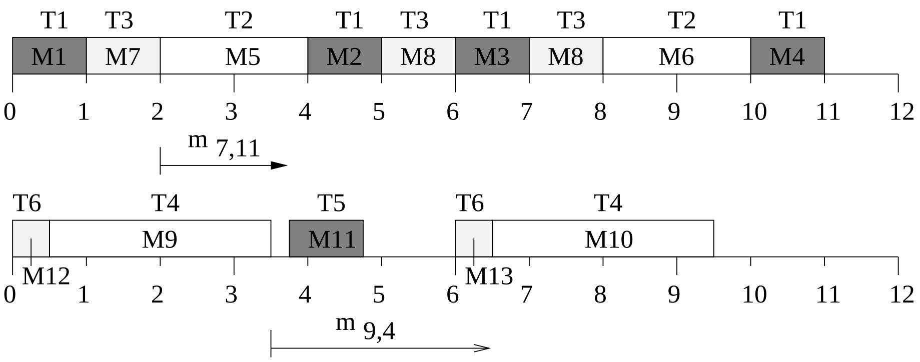
Figure 4: The branching function.

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 $E_j(\gamma_C) - D_j = E_j(\gamma_V) - D_j = lateness(M_{last_m})$. 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_{last_m} as the latest task in the new schedule, γ_C. This guarantees progress. Figure 5 illustrates branching in Case 3. It shows the schedule obtained by branching from vertex $Child_2$ shown in Figure 3 by letting the predecessor M_7 inherit the lateness of the latest module M_{10}. The new deadline of M_7 becomes $D_7 = 5 - 1.25 = 3.75$. Intuitively, M_7 must complete by that deadline in order for M_{10} to complete in time. The resulting new EDF schedule shifts M_7 earlier (in accordance with its new deadline) which happens to result in a globally optimal schedule. Figure 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 $Sync(V)$ (i) all the mutual exclusion constraints, and (ii) all precedence constraints among modules on different processors, then computing vertex cost subject to the remaining set of local precedence constraints, say $Sync_{rem}(V) \subset Sync(V)$, 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 $Sync_{rem}(V)$.)

The lateness of the computed EDF schedule is globally optimal among all schedules that satisfy $Sync_{rem}(V)$. This is because (i) since no exclusion constraints are present in $Sync_{rem}(V)$ EDF is



$Ord(V) = m_{7,11}, m_{9,4}$

$Sync(V) = \{M_7 \; precedes \; M_{11}, M_7 \; precedes \; M_8, M_9 \; precedes \; M_4, M_9 \; excludes \; M_{11}, M_{11} \; precedes \; M_{10}\}$

Figure 5: Branching in Case 3.

locally optimal, and (ii) since all constraints in $Sync_{rem}(V)$ 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 L_{min}.

Finally, since any solution $S \in Space(V)$ to the original task scheduling problem satisfies the constraint set $Sync(V)$, it satisfies, by implication, the constraint subset $Sync_{rem}(V)$. Thus, the lateness of S cannot be less than the global optimum L_{min}. Thus, L_{min} 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.

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 4 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

branch()

if *Case* 1 **return** *solution*(V) optimal

if *Case* 2

Find $M_i, M_j \in B_{last_m}$ where $D_i > D_{last_m}$ and $\exists\{M_i\ excludes\ M_j\} \in Sync(V)$

Generate Child C_1 where $Sync(C_1) = Sync(V) - \{M_i\ excludes\ M_j\} + \{M_i\ precedes\ M_j\}$

Generate Child C_2 where $Sync(C_2) = Sync(V) - \{M_i\ excludes\ M_j\} + \{M_j\ precedes\ M_i\}$

if *Case* 3

if $\exists m_{j,i}$ where $M_i \in B_{last_m}$, and M_j, M_i run on different processors

Increase the priority of $m_{j,i}$ as shown in Figure 4

Let $Sync(C) = Sync(V)$

else $\forall\ M_j$ where $M_j\ precedes\ M_i \in Sync(V)$, $M_i \in B_{last_m}$ and M_j, M_i run on different processors

Generate Child C with $Sync(C) = Sync(V)$ and $D_j = \min(D_j, E_j + E_{last_m} - D_{last_m})$

Figure 6: The branching function.

given below, each point is obtained by averaging 25 readings.

Figure 7 shows the effect of concurrency. The number of concurrent application threads per processor is varied from 1 to 9. Note that the total number of application threads may be much more than the above. By concurrent threads we mean only those that become ready to execute during the same time intervals. CPU utilization is fixed at 90%, 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 6 threads per processor or less, the average number of generated vertices is less than 5. 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.

Figure 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 8. The algorithm performs very well for utilization up to 80%. 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

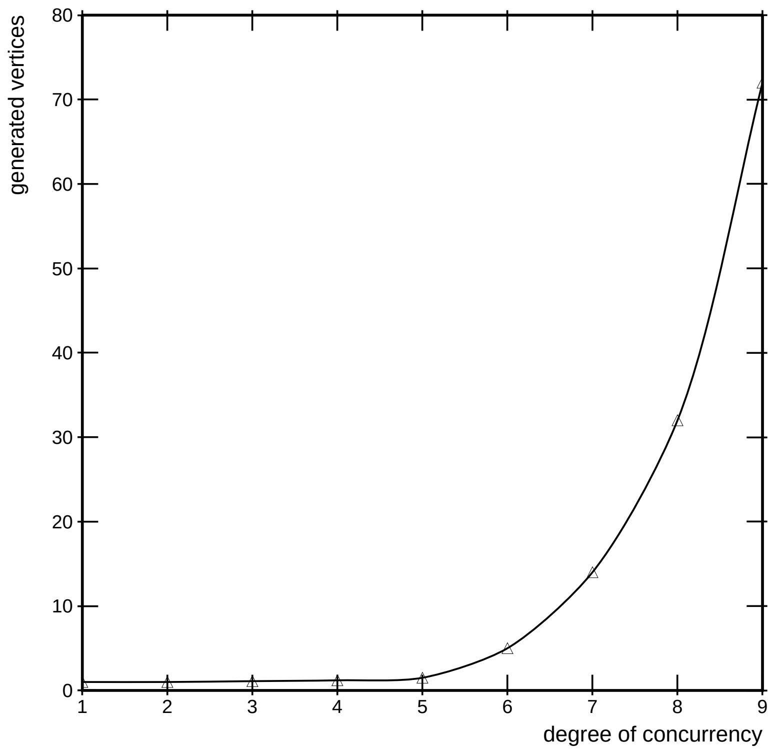


Figure 7: Effect of application concurrency.

length of the busy period, and therefore the increase in branching factor.

Finally, Figure 9 illustrates the effect of module interaction measured in number of communicated messages within the system. CPU utilization was fixed at 90%, and the degree of concurrency was fixed at 8. 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

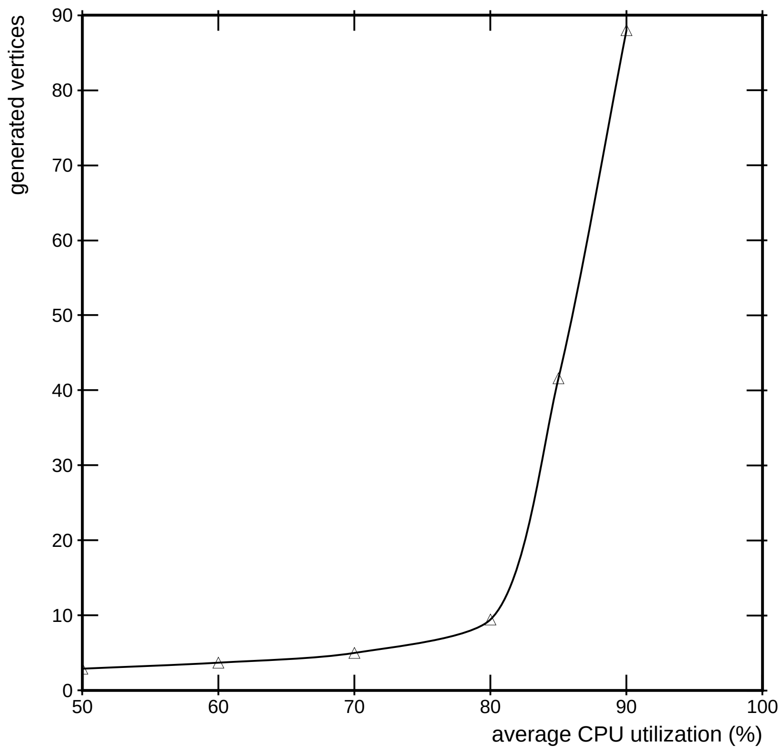


Figure 8: Effect of processor utilization.

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 30 randomly-generated cases of periodic task sets with 90% 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

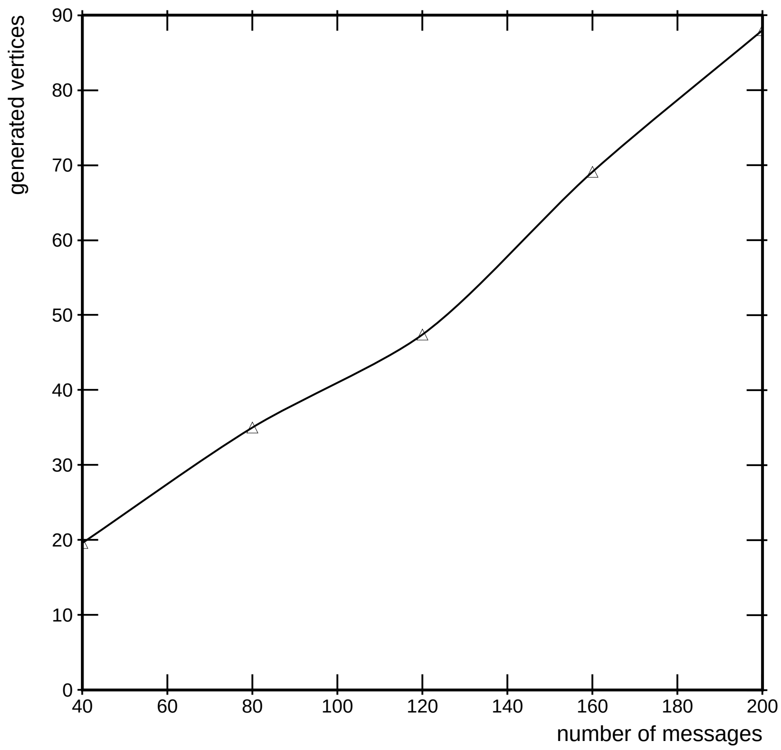


Figure 9: Effect of module interaction.

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% less 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.

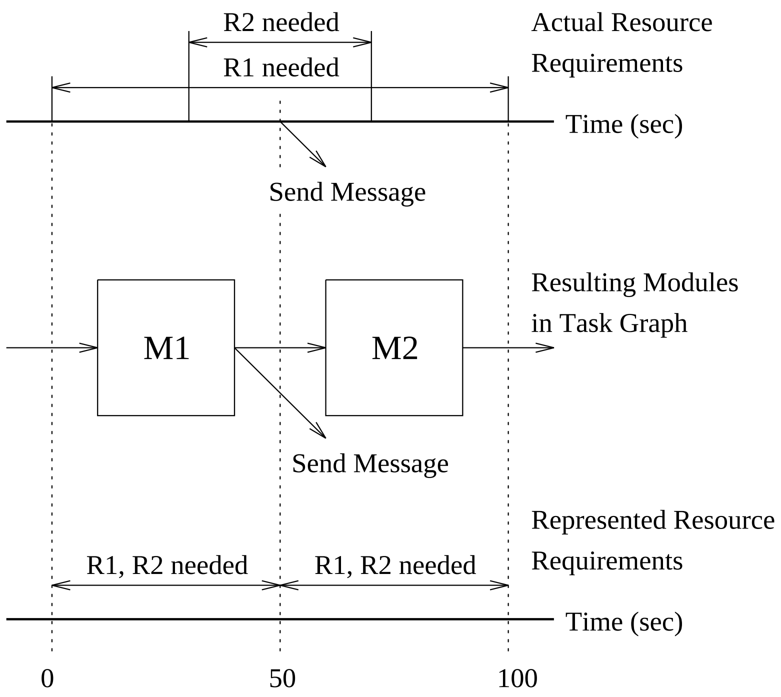


Figure 10: Accounting for contention.

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.

Figure 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 Figure 10 the exclusion constraints do not prevent the scheduler from inserting a module which uses R1 or R2 between M_1 and M_2, which should not be the case. In wath follows, we present a more realistic version of resource constraints, and describe how the algorithm in Section 2 can be modified to accommodate it.

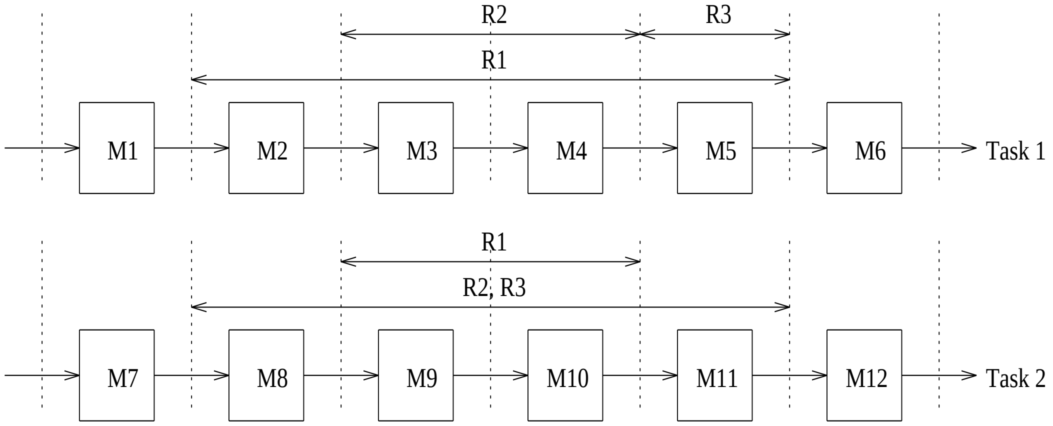


Figure 11: An example of mutual exclusion.

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 $M_1\ M_2\ \cdots\ M_k$ is a module sequence in which M_1 *precedes* M_2 $\cdots$ *precedes* M_k. Thus, the constraints have the general form $ModuleString_1$ *excludes* $ModuleString_2$, meaning that all modules in one of the strings have to terminate before any module in the other can start. For example, Figure 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* $M_9\ M_{10}$ (because of R1)
- C2: $M_3\ M_4$ *excludes* $M_8\ M_9\ M_{10}\ M_{11}$ (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 Figure 11 a potential deadlock arises because $task_1$ and $task_2$ lock resources R1 and R2 in different orders.

5.2 Algorithm Modifications

A close look to 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* $M_9\ M_{10}$ can simply be viewed as a simple exclusion constraint M_a *excludes* M_b, where $M_a = M_2 M_3 M_4 M_5$, and $M_b = M_9 M_{10}$. The algorithm can account for such constraints as discussed earlier. Our only concern is to avoid deadlocks when computing a valid solution, $Solution(V)$. 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 $Solution(V)$ 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 $Solution(V)$, it is circumvented using a similar technique. Consider a deadlock that occurs during the computation of a solution schedule $Solution(V)$ 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 procede. 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. $Solution(V)$ 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 and 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 space in a way to reduce the maximum task lateness.

[5]The replacement of an exclusion constraint by complementary precedence constraints has been described in Section 2.2.3.

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.

References

[1] D. D. Kandlur, K. G. Shin, and D. Ferrari, "Real-time communication in multi-hop networks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1044–1056, October 1994.

[2] J. Xu and D. L. Parnas, "Scheduling processes with release times, deadlines, precedence, and exclusion relations," *IEEE Trans. Software Engineering*, vol. SE-16, no. 3, pp. 360–369, March 1990.

[3] T. Shepard and M. Gagne, "A pre-run-time scheduling algorithm for hard real-time systems," *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 669–677, Jul 1991.

[4] T. F. Abdelzaher and K. G. Shin, "Comment on 'a pre-run-time scheduling algorithm for hard real-time systems," *IEEE Transactions on Software Engineering*, vol. 23, no. 9, pp. 599–600, September 1997.

[5] J. Xu, "Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations," *IEEE Transactions on Software Engineering*, vol. 19, no. 2, pp. 139–154, February 1993.

[6] D.-T. Peng and K. G. Shin, "Optimal scheduling of cooperative tasks in a distributed system using an enumerative method," *IEEE Trans. Software Engineering*, vol. 19, no. 3, pp. 253–267, Mar 1993.

[7] R. Agne, "A distributed offline scheduler for distributed hard real-time systems," in *Distributed Computer Control Systems. Proceedings of the 10th IFAC Workshop*, pp. 35–40, Summering, Austria, September 1991.

[8] K. Jeffay, "On latency management in time-shared operating systems," in *Real-Time Operating Systems and Software*, pp. 86–90, May 1994.

[9] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *Journal of Real-Time Systems*, vol. 2, no. 3, pp. 181–194, September 1990.

[10] M. D. Natale and J. A. Stankovic, "Dynamic end-to-end guarantees in distributed real-time systems," in *Proc. Real-Time Systems Symposium*, pp. 216–227, December 1994.

[11] K. Ramamritham, "Allocation and scheduling of complex periodic tasks," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 108–115, 1990.

[12] D.-T. Peng, K. G. Shin, and T. F. Abdelzaher, "Assignment and scheduling of communicating periodic tasks in distributed real-time systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 12, pp. 745–758, December 1997.

[13] W. H. Kohler and K. Steiglitz, "Enumerative and iterative computational approach," *Computer and Job-Shop Scheduling Theory*, pp. 229–287, 1976.

[14] M.-I. Chen and K.-J. Lin, "Dynamic priority ceilings: A concurrency control protocol for real-time systems," *Journal of Real Time Systems*, vol. 2, no. 4, pp. 325–346, 1990.