Replication and Allocation of Task Modules in Distributed Real-Time Systems

Chao-Ju Hou
Computer Engineering Division
Dept. of Elect. and Computer Eng.
The University of Wisconsin
Madison, WI 53705-1691
jhou@ece.wisc.edu

Kang G. Shin
Real-Time Computing Laboratory
Dept. of Elect. Eng. and Comp. Science
The University of Michigan
Ann Arbor, MI 48109-2122
kgshin@eecs.umich.edu

ABSTRACT
This paper addresses the problem of replicating and allocating periodic task modules to processing nodes (PNs) in distributed real-time systems subject to task precedence and timing constraints. The probability that all tasks can be completed before their deadlines — termed as the probability of no dynamic failure (PND) — is used as the performance-related reliability measure. Modules which are critical in meeting task deadlines are then selected using the critical path analysis. To provide the timing correctness embedded in PND, both original and replicated task modules are not only assigned to PNs, but also scheduled on each PN so as to meet the deadlines of all tasks.

The module allocation scheme uses (1) the branch-and-bound method to implicitly enumerate all possible allocations while effectively pruning unnecessary search paths; and (2) the module scheduling scheme to schedule the modules assigned to each PN. Several numerical examples are presented to illustrate the proposed scheme.

1 Introduction
There has been an increasing need of timely and dependable services for such embedded real-time systems as aircraft, intelligent vehicles, automated factories, and industrial process controls. Such applications are usually realized by executing a number of cooperating/communicating tasks before their deadlines imposed by the corresponding mission/function. The availability of inexpensive, high-performance processors and high-capacity memory chips has made distributed computing systems a natural candidate for the realization of these real-time applications.

One can make the execution of both periodic and aperiodic tasks not only logically correct but also completed before their deadlines by (1) partitioning periodic tasks into a set of communicating modules, (2) statically allocating these modules (and possibly their replicas) to processing nodes (PNs) in a distributed system, and (3) dynamically distributing aperiodic tasks as they arrive according to the load state of each PN.

Partitioning tasks is usually treated as an adaptive load sharing problem. Both of these are not the intent of this paper; see [1] for an example of partitioning real-time tasks, and see [2, 3, 4] for examples of dynamic load sharing in distributed real-time systems. In this paper, we consider instead the issue of replicating and allocating periodic task modules to PNs in a distributed system so as to fully utilize the inherent parallelism, capacity, and reliability of the system.

The problem of allocating tasks/modules in a distributed system has been studied by many researchers with respect to different objective functions. These objective functions can be roughly grouped into four categories: (O1) minimization of total computation and communication times in the system; (O2) load balancing by minimizing the statistical variance of processor utilization; (O3) minimization of maximum computation and communication times on a PN, the objective function of which was termed the maximum turnaround time in [11], the bottleneck processor time in [12, 13], and the system hazard in [14]; (O4) maximization of the reliability function of both PNs and communication links [15].

O1 and O2 are suitable for a distributed system executing multiple simultaneous non real-time applications, where maximizing the total throughput or minimizing the average response time is the main concern. However, for real-time systems, the timing correctness of each individual task must be considered, because failure to correctly complete a task in time could cause a catastrophe. Thus, O3, which is based on the worst-case behavior, is more suitable for assessing the timeliness of real-time systems. In this paper, we use the probability, PND, that all tasks within a planning cycle are completed before their deadlines (which was termed in [16] as the probability of no dynamic failure) as the objective function. The planning cycle is the time period within which the task-invocation behavior repeats itself throughout the entire mission, and thus completely specifies the entire task system. How to incorporate O4 into the probability of dynamic failure has been treated in [17].

Module allocation schemes must be equipped with the ability to tolerate node failures by allocating replicated modules to distinct PNs. The outputs (or the completion notifications) of the replicas of a module are sent to all of its successor modules. A module is enabled when at least one replica of each of its predecessors is completed. If transient or permanent
faults occur to a PN, the replicas of all the modules assigned to this PN continue to be executed on some other healthy PNs so that the subsequent modules can be completed in time. When replicating modules in order to maximize $P_{ND}$ and improve system reliability, one must consider (1) which modules to be replicated, e.g., those modules whose completion is critical to the timely completion of tasks; (2) how many copies of each selected module, i.e., the number of copies needed for each critical module should be determined by system capacity, the minimum $P_{ND}$ that should be guaranteed, and the degree of fault-tolerance achieved; (3) the assignment and scheduling of the replicas on PNs. Our objective is to allocate replicated modules to distinct PNs to provide a guaranteed $P_{ND}$ in the presence of node failures while fully utilizing the inherent parallelism and capacity of the system.

We first model the task system with a task flow graph (TG) which describes computation and communication modules as well as the precedence constraints among them. Second, we use the critical path analysis to determine the modules that are critical in completing tasks in time, and hence, should be replicated. Then, we use the module replication and allocation scheme to determine the optimal number of each critical module’s replicas subject to a pre-specified $P_{ND}$ value, and to search for the optimal module allocation for all original and replica modules. The computational complexity is reduced by deriving an upper bound of the objective function with which we determine whether to expand or prune intermediate vertices (corresponding to partial allocations) in the state-space search tree. On the other hand, because of the timing aspects embedded in the objective function, the performance of any resulting assignment strongly depends on how the assigned tasks/modules are scheduled. Thus, when we evaluate an upper-bound (exact) objective function for a partial (complete) allocation, we use a module scheduling scheme (with polynomial time complexity) to schedule all the modules assigned to a PN so as to minimize the maximum tardiness of modules subject to precedence constraints.

Ramanritham [18] used a heuristic-directed search technique with tunable design parameters to (1) determine whether or not a group of communicating modules should be assigned to the same PN, and (2) allocate different groups of modules to PNs and schedule them with respect to their latest-start-times and precedence constraints. As compared to this work, we use a finer granularity in modeling the real-time task system. For example, we include probabilistic branches/loops in task graphs and allow communications between periodic tasks. Although Ramanritham also considered fault-tolerance via module replication, the degree of replication is pre-determined in an ad hoc manner without any rigorous justification. By contrast, we focus on module replication and allocation in a well-defined analytic framework with $P_{ND}$ as the objective function.

The rest of the paper is organized as follows. In Section 2, we describe how to model real-time task systems. Assumptions on the distributed system are also stated there. In Section 3, we discuss how to determine the modules that should be replicated by using the critical path analysis. Section 4 describes our module replication and allocation scheme. The objective function $P_{ND}(x)$ is derived in Section 5. Section 6 presents demonstrative examples, and the paper concludes with Section 7.

2 Task and System Models

2.1 The Task System

Real-time tasks are either periodic or non-periodic. A periodic task is invoked at fixed time intervals and constitutes the base load of the system. Its attributes, such as the required resources, the execution time, and the invocation period, are usually known a priori. A non-periodic task, on the other hand, is invoked randomly in response to environmental stimuli, especially to unanticipated abnormal situations. The main intent of this paper is to address the problem of replicating and allocating the modules of periodic tasks.

Planning cycle: To analyze the behavior of periodic tasks, we only need to consider the task behaviors within a specific period, the task behaviors during which will repeat for the entire mission lifetime. Such a period is called the planning cycle of periodic tasks and is defined as the least common multiple (LCM) $L$ of $(p_i : i = 1, 2, \ldots, N_T)$, where $p_i$ is the period of a task $T_i$ and $N_T$ is the total number of periodic tasks in the system. That is, the planning cycle is the time interval $[t_0 + kL, t_0 + (k + 1)L]$, where $t_0$ is the mission start time, and $k$ is a nonnegative integer.

Attributes and precedence constraints among modules: Each task can be decomposed into smaller units, called modules. Each module $M_i$ requires $e_i$, units of execution time. The execution time of a module could be its worst-case execution time or its exact execution time if known. Since extensive simulations and testing are required before putting any critical real-time system in operation (e.g., fly-by-wire computers), the system designer is assumed to have a good, albeit sometimes incomplete, understanding of either the exact or the worst-case execution time of each module.

The execution order of modules imposes precedence constraints among them. These precedence constraints are of the form $M_i \rightarrow M_j$, meaning that the completion of $M_i$ of a task enables another module $M_j$ of the same task to be ready for execution. On the other hand, tasks communicate with one another to accomplish the overall control mission. The semantics of message communication between two cooperating tasks also impose precedence constraints between the associated modules of these tasks. This kind of precedence constraints is also of the form $M_i \rightarrow M_j$, except that $M_i$ and $M_j$ now belong to different tasks.

If $M_i$ and $M_j$ are assigned to the same PN, communication between them can be achieved via accessing shared memory. Overheads of such communications are usually much smaller than those when $M_i$ and $M_j$ reside on different PNs. Any two communicating modules that reside on two different PNs will incur interprocessor communication (IPC) which requires extra processing such as packetization and depacketization. IPC introduces a communication delay which is a function of intermodule communication (IMC) volume (measured in data units) and the link delay between the two communicating PNs.

Task Flow Graph (TG): A TG is commonly used to de-
scribe the logical structure of modules, and the communications and precedence constraints among them. A TG is composed of four types of subgraphs: chain, AND-subgraph, OR-subgraph, and loop. See [1, 17] for a detailed account of the four component subgraphs. Here we assume that the probability for taking a particular branch in an OR-subgraph or for repeating/exiting the body of a loop is independent of that for others. These probability values could be set to the worst-case values and can be obtained from the extensive simulations and testing — usually required of critical real-time systems — during the system-design phase. Fig. 1 (a) shows a simple example of a TG.

Communication primitives: The semantics of the most general communication primitive, SEND-RECEIVE-REPLY, can be embedded into precedence relations between modules. If module $M_a$ of task $T_i$ issues a SEND to task $T_j$, $T_i$ remains blocked, or cannot execute module $M_e$ that follows $M_a$, until the corresponding REPLY from $T_j$ is received. If the module, $M_c$, responsible for the corresponding communication activity on $T_j$'s side executes a RECEIVE before the SEND arrives, $T_j$ also remains blocked. For example, the communication activities between tasks in Fig. 1 (a) can be embedded into the precedence constraints between modules as shown in Fig. 1 (b).

2.2 The Distributed System

As discussed in Section 1, fault-tolerance can be achieved by allocating module replicas to distinct PNs. However, it might be intractable to replicate all the modules due to limited system resources. Moreover, it might not be necessary to replicate the modules that are not subject to stringent time requirements and can tolerate the worst-case recovery delay. That is, if the PN fails before or when some less time-critical modules are executed, we may employ, depending on the fault type, different methods, like retry, checkpoint, rollback recovery, and component replacement, to tolerate faults at the cost of recovery/switching time overhead. Modules which can tolerate such a recovery delay need not be replicated.

Specifically, let $r_i$ be the earliest release time when $M_i$ can start its execution, $LC_i$ be the latest completion time of $M_i$, and $DL_i$ be the deadline of $M_i$. The time for packetization and depacketization is lumped into module execution time.

Figure 1: An example of task flow graph.
to ensure that all of its succeeding modules will meet their latest completion times, \( e_i \) be the execution time of \( M_i \), and \( t_{rec} \) be the worst-case recovery time. If the interval between the latest completion time, \( LC_i \), and the earliest release time, \( r_i \), of a module, \( M_i \), is less than or equal to the sum of the execution time, \( e_i \), of \( M_i \), and the worst-case error recovery time \( t_{rec} \), i.e.,

\[
LC_i - r_i \leq e_i + t_{rec}, \tag{3.1}
\]

then \( M_i \) cannot be completed in time (even with the use of recovery/replacement methods) in case of a node failure, and thus, should be replicated.

The key step lies in how to compute \( LC_i \) and \( r_i \) for each module \( M_i \). We use the critical path approach to calculate (1) \( r_i \), from the invocation time of the task to which \( M_i \) belongs and which precedes \( M_i \), and (2) \( LC_i \), from the deadline of the task to which \( M_i \) belongs and which succeeds \( M_i \). Specifically, we first "transform" the TG which contains probabilistic branches/loops into a deterministic one by replacing (1) an OR-subgraph with the corresponding AND-subgraph (i.e., ignoring branching probabilities), and (2) a loop with the cascaded \( n_i \) copies of its loop body, where \( n_i \) is its maximum loop count. Second, we number all modules in the (transformed) TG in acyclic order such that if \( M_i \) \( \rightarrow \) \( M_j \) then \( i < j \). Then, we use the critical path approach to calculate \( LC_i \) and \( r_i \). Let \( LC_i \) be initially set to the deadline of the task to which \( M_i \) belongs. Then, modify \( LC_i \) as

\[
LC_i = \min \{ LC_i, \min \{ LC_j - e_j : M_i \rightarrow M_j \} \}, \quad i = N-1, \ldots, 1, \tag{3.2}
\]

where \( N \) is the number of original modules to be allocated within a planning cycle. Note that Eq. (3.2) computes backward from \( i = N-1 \) to \( i = 1 \), because \( M_N \) has no successor by the nature of acyclic order, and thus, the latest completion time of \( M_N \) is exactly the deadline of the task it belongs to. Similarly, the earliest release time, \( r_i \), of \( M_i \) is obtained by initially setting \( r_i \) to the invocation time of the task to which \( M_i \) belongs. Then, modify \( r_i \) as

\[
r_i = \max \{ r_i, \max \{ r_j + e_j : M_j \rightarrow M_i \} \}, \quad 2 \leq i \leq N, \tag{3.3}
\]

where \( r_i \) is the invocation time of the task to which \( M_i \) belongs. All the modules that are subject to the same timing constraints and satisfy Eq. (3.1) form a critical path. Note that we do not consider the possible IPC communication time between two modules, because (1) \( t_{rec} \) is assumed to be much larger than the IPC time and thus dominate in the expression of Eq. (3.1), and (2) sequentially-executing modules subject to the same tight timing constraints (and thus lie on a critical path) tend to be allocated to the same PN by the allocation scheme [17], and thus no IPCs are incurred on the critical path. Also implied in Eqs. (3.1)-(3.3) is that at most \( t_{rec} \) units of time can be "delayed" along any non-critical execution path from an entry point to an end point in the TG. The modules that satisfy Eq. (3.1) are then selected as the modules that should be replicated. Each module replica inherits the same execution time and timing/precedence constraints as its original.

Fig. 2 shows an example of how \( r_i \)'s and \( LC_i \)'s are calculated in the TG given in Fig. 1 (b). For example, the release time

\[
r_4 = \max \{ r_4, r_3 + e_1, r_2 + e_2, r_5 + e_3 \}
\]

and the latest completion time, \( LC_{12} \), of \( M_{12} \) is calculated as

\[
LC_{12} = \min \{ LC_{12}, LC_{13} - e_{13} \} = \min \{10, 12 - 1 \} = 10.
\]

The execution path \( M_2 \rightarrow M_3 \rightarrow M_4 \rightarrow M_5 \rightarrow M_9 \rightarrow M_{10} \rightarrow M_{11} \rightarrow M_{12} \rightarrow M_{13} \) is critical with respect to \( T_3 \)'s timing constraint, cannot tolerate any recovery delay, and thus all the modules on this path should be replicated.

### 4 Module Replication and Allocation Scheme

The module replication and allocation problem can be formulated as that of maximizing \( P_{ND}(x) \) over all possible allocations subject to

\[
\sum_{k=1}^{K} x_{ik} = 1, \quad \text{for} \ i \in \text{modules not replicated}, \quad \quad \sum_{k=1}^{K} x_{ik} = m_m, \quad \text{for} \ i \in \text{modules replicated},
\]

where \( x_{ik} = 1 \) if and only if \( M_i \) is assigned to \( N_k \), and \( m_m \) is the number of replicas yet to be determined. The precedence constraints among modules are figured in the calculation of module release times and latest completion times, and the timing constraints on modules/tasks are considered.
when \( P_{ND}(z) \) is evaluated; for example, \( P_{ND}(z) = 0 \) if some of the tasks miss their deadline under \( z \). The expression for \( P_{ND}(z) \) will be derived in Section 5.

We use a module allocation (MA) scheme to solve the above formulated problem with a determined value of \( m_m \). To get the optimal assignment and scheduling for both original and replica modules, and the corresponding optimal objective value \( P'_{ND} \), the MA scheme uses: (1) the branch-and-bound (BB) method to implicitly enumerate all possible allocations while effectively pruning unnecessary paths in the search tree; and (2) the module scheduling (MS) scheme to schedule the modules assigned to each PN subject to the precedence constraints and the latest module completion times. The description and analysis of MS will be given in Section 5.1.

Also, since all replicas of a module inherit the same precedence and timing constraints, and are hence subject to the same source, \( z \), and scheduling of all modules in the augmented task graph, TG, are numbered in acyclic order. The scheme begins with a null allocation \( z_0 \) which corresponds to the root of the search tree, and allocates modules in the order of their acyclic numbering. Let TG(z) denote the set of modules which are already allocated under \( z \), and \( AN \) the set of active vertices in the search tree to be considered for expansion. \( AN \) is determined by the bounding test. Expanding a vertex \( z \in AN \) corresponds to allocating the module, \( M_i \), with the smallest acyclic number in TG \( z \) to a PN, where \( N \) denotes the difference of two sets. Only those PNs which survive the branching test will be considered as candidates for allocating \( M_i \). The dominance relation used in the branching test is as follows. \( M_i \) can be invoked after all its precedence constraints are met and must be completed by its latest completion time, \( LC_i \), to ensure that all its succeeding tasks meet their deadlines. Hence, if (1) the idle time of a PN, say \( N_i \), during the interval \( [r_i, LC_i] \) is smaller than \( c_i \) and (2) the module, say \( M_j \), scheduled to be executed last on \( N_i \) in \( [r_i, LC_i] \) under a partial allocation \( z \) has tighter timing constraints than \( M_i \) (so no preemption on \( N_i \) to ensure the completion of \( M_i \) before \( LC_i \)), then allocating \( M_i \) to \( N_i \) is likely to miss \( M_i \)'s latest completion time. Thus, \( N_i \) should not be a candidate PN for allocating \( M_i \), i.e., fails the branching test.

The bounding test is then applied to those vertices expanded from \( z \) by allocating \( M_i \) to one of the candidate PNs. The UBOF, \( P'_{ND}(y) \), of each newly-generated (intermediate) vertex \( y \) is calculated by scheduling modules \( z \in TG(y) \) with the MS scheme described in Section 5.1 and evaluating \( P_{ND}(y) \) with the expression derived in Section 5.2. If a vertex \( y \) has its \( P_{ND}(y) \) greater than the currently best objective function value \( P'_{ND} \), it survives the bounding test, might possibly lead to the optimal solution, and will be made active (i.e., put into \( AN \)) and considered for vertex expansion in the next stage; otherwise, it will be pruned. The algorithm terminates when an optimal solution is found.

The branching and bounding tests used to achieve BB efficiency were treated in [17]. The interested readers are referred to [17] for a detailed account of them. The MA scheme is outlined below.

**MA Scheme:**

**Step 1.** Generate the root, \( z_0 \), of the search tree, which corresponds to a null allocation. Set \( AN := \{ z_0 \} \).

**Step 2.** Set TG(\( z_0 \)) := \( \emptyset \), \( z_{opt} := z_0 \), and the objective function value achieved by \( z_{opt} \), \( P_{ND} = 0.0 = P_{ND}(z_0) \).

**Step 3.** While \( AN \neq \emptyset \) do

/* an optimal allocation has not yet been found */

\[\text{TG}(x) = \text{TG} \text{ if } x \text{ is a complete allocation.}\]
Step 3.1. Node Selection Rule:
Step 3.1.1. Select the vertex \( x \in AN \) with the largest \( P_{ND}(z) \).
Step 3.1.2. If \( P_{ND}(z) < P_{ND}^* \), terminate MA, and \( z_{opt} \) is the optimal solution. Otherwise, set \( M_i \) to be the module \( \in TG \setminus TG(z) \) with the smallest acyclic number, and \( AN := AN \setminus \{x\} \).

Step 3.2. Branching Test:
Step 3.2.1. Conduct the branching test on each PN. Only those PNs which survive the branching test will be considered for allocating \( M_i \).
Step 3.2.2. Expand \( z \) by generating its valid child vertices, each of which corresponds to allocating \( M_i \) to one of the surviving PNs.

Step 3.3. Bounding Test: For each newly generated vertex \( y \).
Step 3.3.1. Use MS to find an optimal schedule for TG(y) under \( y \) and calculate the UBOF, \( P_{ND}(y) \).
Step 3.3.2. If \( P_{ND}(y) \leq P_{ND}^* \), then prune \( y \). Otherwise, the following two cases are considered:

Case 1. If \( y \) is a partial allocation, then set \( AN := AN \cup \{y\} \), i.e., make \( y \) an active vertex.
Case 2. If \( y \) represents a complete assignment, \( P_{ND}(y) \) is the actual \( P_{ND} \) achieved under \( y \). Since \( P_{ND}(y) > P_{ND}^* \), set \( z_{opt} := y \) and \( P_{ND} = P_{ND}(y) \) to indicate that \( y \) has now become the best allocation found thus far.

5 Evaluation of \( P_{ND}(x) \)
We first describe how MS schedules all the modules assigned to a PN, say \( N_k \), under \( x \) to minimize the maximum module tardiness subject to task release times and precedence constraints. By applying MS to each PN, we can obtain a module schedule under \( x \). Second, we calculate the probability \( P(T_i) \) is timely completed under \( x \). \( P_{ND}(x) \) can then be calculated from \( P(T_i) \) is timely completed under \( x \), \( \forall T_i \).

5.1 Module Scheduling Scheme
To facilitate the description and analysis of MS, we introduce the following notations:

- \( TG_c \): a component task graph of TG. If TG contains loops or OR-subgraphs, it will be replaced by a set of component task graphs without loops and OR-graphs before applying MS (to be discussed in Section 5.2).
- \( TG_c(z) \): the set of modules \( \in TG_c \) under \( z \).
- \( S_k(z) = \{ M_i : x_{ik} = 1 \} \): the set of modules assigned to \( N_k \) under \( z \).
- \( C_i \): the completion time of \( M_i \).
- \( f_i(C_i) \): the cost incurred by completing \( M_i \) at \( C_i \).
- \( \bar{e}_i \): the modified execution time of \( M_i \), where
  \[ \bar{e}_i = \begin{cases} e_i & \text{if } M_i \text{ is scheduled upon } r_i \\ C_i - r_i & \text{otherwise.} \end{cases} \]
\( \bar{e}_i \) is used to include the effect of queueing \( M_i \) on the release times of modules that succeed \( M_i \).

- \( \text{com}_{ij}(x) \): the IMC time bet. \( M_i \) and \( M_j \) under \( x \).
- \( d_{ij} \): the IMC volume (measured in data units) between \( M_i \) and \( M_j \).
- \( t_{mn} \): the link delay (measured in time units per data unit) of link \( t_{mn} \).
- \( n(k, \ell) \): the number of edge-disjoint paths between \( N_k \) and \( N_{\ell} \).
- \( I(m, n, k, \ell) \): the indicator variable such that \( I(m, n, k, \ell) = 1 \) if \( t_{mn} \) lies on one of the \( n(k, \ell) \) edge disjoint paths between \( N_k \) and \( N_{\ell} \).
- \( Y_{m,n,k,\ell} = \frac{1}{n(k, \ell)} \sum_{m, n=1}^{\infty} \sum_{k, \ell=1}^{\infty} I(m, n, k, \ell) \cdot t_{mn} \): the delay (in time units per data unit) bet. \( N_k \) and \( N_{\ell} \).
- \( B \): the minimal set of modules that are processed without any idle time in \( [r(B), c(B)] \), where \( r(B) = \min_{M_i \in B} r_i \), \( c(B) = r(B) + e(B) \), and \( e(B) = \sum_{M_i \in B} e_i \).
- \( d_{G,i} \): the outdegree of \( M_i \) within a block of modules under consideration.

Specifically, \(|S_k(x)|\) modules (possibly belonging to different tasks) are to be scheduled preemptively on \( N_k \). Each module \( M_i \) becomes available upon its release at time \( r_i \), which is initially set to the invocation time of the task to which \( M_i \) belongs. If \( M_j \rightarrow M_i \), then \( M_i \) cannot start its execution before the completion of \( M_j \), regardless whether \( M_i \) and \( M_j \) are assigned to the same PN or not. Execution of a module may be preempted and then resumed later. Associated with each \( M_i \) is a monotone nondecreasing cost function \( f_i(C_i) \).

We want to find a schedule for the modules in \( S_k(x) \) such that \( f_{max}(S_k(x)) \leq f_{max}(S_k(x)) f_i(C_i) \) is minimized. The schedule with the minimal cost \( f_{max}(S_k(x)) \) is said to be an optimal schedule of \( S_k(x) \).

Before proceeding to describe and analyze MS, we define the cost function \( f_i(C_i) \):

\[
f_i(C_i) = C_i - LC_i, \tag{5.1}
\]
where \( LC_i \) is the latest completion time of \( M_i \) with communication times considered now, and \( C_i \) is the completion time of \( M_i \) determined by MS. If \( C_i > LC_i \), a positive cost will occur. Thus, with the definition of this cost function, minimizing the maximum cost function is equivalent to minimizing the maximum tardiness of modules in \( TG_c \).

To obtain \( LC_i \), of \( M_i \in TG_c \), let \( LC_i \) be initially set to the deadline of the task to which \( M_i \) belongs, and then modify \( LC_i \) as

\[
LC_i = \min\{LC_i, \min_{j \neq i} (LC_j - \bar{e}_j - \text{com}_{ij}(x) : M_i \rightarrow M_j)\},
\]

\[ i = N_c - 1, \ldots, 1. \tag{5.2} \]
where the modules are numbered in acyclic order, \( N_e \) is the number of modules in \( TG_e \), and

\[
com_{ij}(x) = \begin{cases} 
0, & \text{if } M_i \to N_k \text{ and } M_j \to N_k \text{ under } x, \\
d_{ij} y_{kr}, & \text{if } M_i \to N_k \text{ and } M_k \to N_l \text{ under } x.
\end{cases}
\]

When \( x \) is a partial allocation and either \( M_i \) or \( M_j \) or both have not yet been assigned, \( com_{ij}(x) \) is (optimistically) assumed to be 0.

To obtain \( r_i \), of \( M_i \in TG_e(x) \), let \( r_i \) be initially set to the invocation time of the task to which \( M_i \) belongs, and then modify \( r_i \) as

\[
r_i = \max \{r_i, \max \{\tau_j + \tilde{\epsilon}_j + com_{ij}(x) : M_j \to M_i\}\},
\]

\[2 \leq i \leq N_{ce},\] (5.3)

where \( r_i \) is the invocation time of the task to which \( M_i \) belongs, and \( \tilde{\epsilon}_j = \max \{C_j - \tau_j, \tilde{\epsilon}_j\} \) is the modified execution time which equals the sum of \( M_j \)'s execution time, \( \epsilon_j \), and \( M_i \)'s queuing time (if \( M_i \) is not scheduled to be executed upon its release). \( \epsilon_j \) is used to include the effect of queuing \( M_i \)'s preceding module, \( M_j \), on \( M_i \)'s release time.

Note that the modified execution times of all \( M_i \)'s preceding modules must be available prior to the calculation of \( r_i \). This is achieved by allocating the modules in the order of their acyclic numbers. When an intermediate vertex \( y \) survives the bounding test and is put in \( AN \), all modules in \( TG_e(y) \) would have been scheduled and their completion times (and thus modified execution times) would have been determined in the bounding process in the previous stage (Step 3.3 in the MA scheme in Section 4). Thus, when \( x \) is expanded from its parent vertex \( y \) by adding the new assignment of \( M_i \), the schedules, completion times and modified execution times of all preceding modules of \( M_i \) must have been determined. So, all the \( \tilde{\epsilon}_j \)'s needed in Eq. (5.3) are known at the time of calculating \( r_i \).

Now, we describe MS, the theoretical base of which is grounded on the result of [21]. First, we arrange the modules \( E_s(x) \) in the order of nondecreasing release times. We then decompose \( E_s(x) \) into blocks, where a block \( B \subset E_s(x) \) is defined as the minimal set of modules processed without any idle time from \( r(B) = \min_{M_i \in B} r_i \) until \( c(B) = r(B) + c(B) \), where \( c(B) = \sum_{M_i \in B} \epsilon_i \). That is, each \( M_i \notin B \) is either completed no later than \( r(B) \) or not released before \( c(B) \).

Obviously, scheduling modules in a block \( B \) is irrelevant to that in other blocks, so we can consider each block separately. Let \( d_i \) denote the outdegree of \( M_i \), within \( B \), i.e., the number of modules \( M_j \in B \) such that \( M_i \to M_j \). For each block \( B \), we first determine the set \( \hat{B} \triangleq \{ M_i : M_i \in B, d_i = 0 \} \), i.e., modules without successors in \( B \), and then select a module \( M_m \) such that

\[
f_m(c(B)) = \min_{M_i \in B} f_i(c(B)),
\] (5.4)

i.e., \( M_m \) has no successor within \( B \) and incurs a minimum cost if it is completed last in \( \hat{B} \). (In case of a tie, we choose the module with the largest acyclic number.) Now, consider an optimal schedule for the modules in \( B \) subject to the restriction that \( M_m \) is processed only if no other module is waiting to be processed. This optimal schedule consists of two parts:

**Sched1:** An optimal schedule with the cost \( f_{max}(B - \{M_m\}) \) for the set \( B - \{M_m\} \) which could be decomposed into a number of subblocks \( \hat{B}_1, \hat{B}_2, \ldots, \hat{B}_g \).

**Sched2:** A schedule for \( M_m \), which is given by \( [r(B), c(B)] - \sum_{M_i \in B} [r(B), c(B)] \), where \( r(B) = \min_{M_i \in B} r_i \), and \( c(B) = r(B) + c(B) \) with \( c(B) = \sum_{M_i \in B} \epsilon_i \).

For this optimal schedule, we have

\[
f_{max}(B) \geq \max\{f_m(c(B)), f_{max}(B - \{M_m\})\} \leq f_{max}(B) \] (5.5)

where the last inequality comes from: (i) \( f_{max}(B) \triangleq \min_{M_i \in B} f_i(c(B)) \geq \min_{M_i \in B} f_i(c(B)) = \min_{M_i \in B} f_i(c(B)) = f_m(c(B)) \) by the way \( \hat{B} \) was constructed from \( B \) and Eq. (5.4); (ii) Since \( B - \{M_i\} \) is a subset of \( B \), \( f_{max}(B) \geq f_{max}(B - \{M_i\}) \), \( \forall M_i \).

It follows from Eq. (5.5) that there exists an optimal schedule in which \( M_m \) is scheduled only if no other module is waiting.
to be scheduled. By repeatedly and recursively applying the above procedure to each of the subblocks \(B_1, B_2, \ldots, B_h\), we obtain an optimal schedule for \(B\). The rationale behind MS is that a PN is never left idle when there are modules ready to execute, and by virtue of the cost function defined, it is always the module \(M_i\) with the smallest \(LC_i\) that will be executed among all released modules.

Fig. 3 gives an illustrative example showing how MS schedules the modules assigned to a PN. \(r_i\) and \(LC_i\), \(1 \leq i \leq 6\), are assumed to have been computed from the entire task graph and are given in the figure. By ordering the modules according to their increasing release times, we obtain processed only when no other modules are waiting since we have \(B_{111} = \{M_1, M_2, M_3, M_4, M_5, M_6\}\) from \([0, 11]\) (i.e., \(r(B_1) = 0\), \(e(B_1) = 11\), and \(c(B_1) = 11\)). Moreover, we have \(B_i = \{M_4, M_5, M_6\}\) and select \(M_6\) to be processed last since \(LC_g > LC_4 > LC_3\). \(B_i - \{M_6\}\) consists of one subblock: \(B_11 = \{M_1, M_2, M_3, M_4\}\) from \([0, 9]\). \(B_11 = \{M_2, M_3, M_4\}\), and we select \(M_3\) to be processed last since \(LC_4 > LC_3 > LC_2\). \(B_111\) consists of one subblock: \(B_111 = \{M_1, M_2, M_3, M_4, M_5\}\) from \([0, 8]\). \(B_1111 = \{M_3\}\), and we select \(M_3\) to be processed last since \(LC_4 > LC_3 > LC_2\). Now \(B_1111 = \{M_3\}\) consists of two subblocks: \(B_1111 = \{M_4, M_5\}\) from \([0, 3]\) and \(B_11111 = \{M_5\}\) from \([4, 6]\). \(B_11112\) itself represents an optimal schedule, since \(B_11112\) consists of a single module. For \(B_1111\), we have \(B_{1111} = \{M_1, M_2\}\) and select \(M_1\) to be processed last since \(LC_1 > LC_2\). The final optimal schedule for \(B_111\) is obtained by combining the optimal schedule of \(M_1\) and \(M_2\). For \(B_1111\), we have \(B_{11111} = \{M_3\}\) and select \(M_3\) to be processed last since \(LC_4 > LC_3 > LC_2\). The final optimal schedule for \(B_{1111}\) is obtained by combining the optimal schedule of \(M_3\) and \(M_4\). The resulting schedule for \(B_1\) is depicted in the last row of Fig. 3.

The MS scheme along with the time complexity in each step is summarized below.

**MS Scheme:**

**Step 1:** Compute the latest completion time \(LC_i\), \(1 \leq i \leq N_c\), of each module in \(TG_c\). This computation requires \(O(N_c^2)\) time.

**Step 2:** Compute the release time \(r_i\) for \(M_i \in TG_c(x)\) with respect to their precedence constraints. This computation, in the worst case, requires \(O(N_c^2)\) time.

**Step 3:** Construct the blocks \(B_1, B_2, \ldots, B_h\) of \(S_k(x)\) for every \(N_k\) by ordering the modules in \(S_k(x)\) according to their nondecreasing release times. This ordering requires \(O(|S_k(x)| \cdot \log |S_k(x)|)\) time, \(\forall k\).

**Step 4:** For each block \(B_i, 1 \leq i \leq h\), update the outdegree, \(d_{gy}\), of every \(M_j \in B_i\). This update requires \(O(|S_k(x)|^2)\) time for all \(B_i \in S_k(x)\).

**Step 5:** For each block \(B_i\), select \(M_m \in B_i\) subject to Eq. (5.4), determine the subblocks of \(B_i - \{M_m\}\), and construct the schedule for \(M_m\) as given in Sched2. Then, update the \(d_{gy}\) of every \(M_j \in B_i - \{M_m\}\) with respect to the subblock of \(B_i - \{M_m\}\) to which \(M_j\) belongs. By repeatedly applying Step 5 to each of the subblocks of \(B_i - \{M_m\}\), one can obtain an optimal schedule. The time complexity for all repeated applications of Step 5 is bounded by \(O(|S_k(x)|^3)\).

Since the time complexity associated with each step is polynomial, the MS scheme is a polynomial scheme.

### 5.2 Calculation of \(P_{NPD}(x)\)

We are now in a position to calculate \(P(T_c\) is timely completed under \(x)\). Conceptually, given \(TG\) and \(x\), we can determine the set, \(S_k(x)\), of modules in \(TG\) assigned to \(N_k\) and then use MS to schedule modules in \(S_k(x)\). The completion time(s) of the last module(s) in \(T_c \cap TG\) under these schedules determines whether \(T_c\) can be completed in time or not. However, since TG may contain loops and/or OR-subgraphs, the release times and the latest completion times of modules needed in Step 3 of MS may not be readily determined. Moreover, one cannot determine which module of \(T_c\) to execute last if the last component in \(T_c\) is an OR-subgraph.

**Component Graphs:** To resolve the above problems, we must eliminate the loops/OR-subgraphs in \(TG\) while retaining all the timing and probabilistic properties of \(TG\). First, we calculate the latest completion time, \(LC\), of \(M_i \in TG\) using Eq. (5.2), assuming that (A1) Every OR-subgraph following \(M_i\), if any, is viewed as an AND-subgraph by ignoring branching probabilities; (A2) Every loop \(L_a\) following \(M_i\), if any, is replaced by a cascade of \(n_{L_a}\) copies of its loop body, where \(n_{L_a}\) is the maximum loop count. With A1 and A2, the \(LC\)'s calculated is the worst-case latest completion time.

Second, we represent each loop \(L_a \in TG\) with the cascaded \(m\) copies of its loop body with probability \((1 - q_a)^m q_a^{m-1}\), where \(1 \leq m \leq n_{L_a}\), and \(q_a\) is the looping-back probability of \(L_a\). The last copy of \(M_i \in L_a\) bears the \(LC\), calculated above, while the \((n_{L-a} - j)\)-th copy of \(M_i\) bears the latest completion time \(LC_i - \sum_e e(L_a)\), where \(e(L_a)\) is the execution time of the loop body. Also, we represent each OR-subgraph \(O_k \in TG\) with its \(n\)-th branch with probability \(q_{o,n}\), where \(1 \leq n \leq n_{O_k}\), \(q_{o,n}\) is the branching probability of the \(n\)-th branch of \(O_k\), and \(n_{O_k}\) is the number of branches in \(O_k\).

The TG can then be represented by the set of all possible combinations — which is termed as the set of component task graphs. For example, if there exists a loop \(L_a\) and an OR-subgraph \(O_k\) in TG, then there are a total of \(n_{L_a} \times n_{O_k}\) component graphs of TG, and with probability \(p_{a,n} = (1 - q_a)^m q_a^{m-1} \cdot q_{o,n}\), the TG is represented by the TG with \(L_a\) replaced by the cascaded \(m\) copies of its loop body and \(O_k\) replaced by its \(n\)-th branch. (One can trivially extend this to the case where there are more than one loop and/or OR-subgraph.)

For each component graph, \(TG_c\), of TG, we then calculate the release time, \(r_i\), of \(M_i \in TG_c\) using Eq. (5.3). Using the \(r_i\)'s and \(LC_i\)'s determined above, we can apply Steps 3-5 in MS to find the best schedules for all modules in \(TG_c\). Note that in a component graph \(TG_c\), the release time, \(r_i\), and the number of times \(M_i\) is executed are both fixed, making it possible to decompose \(S_k(x)\) into blocks.

**Calculation of \(P_{NPD}(x)\):** We now calculate the probability \(P(\text{\(T_c\) is timely completed under \(x)\)}\). In the critical time of \(M_i \in TG_c\), \(D_i\) is defined as the latest time \(M_i\) should be completed for the timely completion of only the task \(T_c\). Note that \(D_i\) can be obtained in the same way as \(LC\), except that the precedence relations, \(M_i \rightarrow M_j\) when \(M_i \not\in TG_c\), are ignored. Therefore, \(D_i\) is initially set to the deadline of \(T_c\) to which \(M_i\) belongs. Then, \(D_i\) is modified as:

\[
D_i = \min(D_i, \min \{D_j - c_j - \text{com}_{ij}(x) : M_i \rightarrow M_j\})
\]
Obviously, \( D_i \geq LC_i \). Also, let

\[
\mathcal{T}_i = \{ M_i : M_i \in T_i \cap TG_c, \delta g_i = 0 \text{ w.r.t. } T_i \cap TG_c \}
\]

be the set of modules without any successor in \( T_i \cap TG_c \). Then, the probability \( P(T_i \text{ is timely completed under } x \in TG_c) \) can be expressed as

\[
P(T_i \text{ is timely completed under } x \in TG_c) = \prod_{M_i \in TG_c} \delta(D_i - C_i),
\]

where \( \delta(\cdot) \) is the step function, i.e., \( \delta(t) = 1 \) for \( t \geq 0 \), and \( \delta(t) = 0 \) otherwise. Consequently,

\[
P(T_i \text{ is timely completed under } x) = \sum_{\{TG_c\}} p_c \cdot P(T_i \text{ is timely completed under } x \in TG_c),
\]

where \( p_c \) is the probability that \( TG \) is represented by \( TG_c \), and \( \{TG_c\} \) is the set of component graphs of \( TG \), and finally

\[
P_{ND}(x) = \prod_{t=1}^{N_T} P(T_i \text{ is timely completed under } x). \tag{5.10}
\]

6 Numerical Examples

We randomly generated both system and task parameters in our numerical experiments. The number of PNs in the distributed system is varied from 3 to 40, and the network topology is arbitrarily generated. The link delay, \( t_{mn} \), associated with \( \ell_{mn} \) is exponentially distributed with mean 0.18, where \( \ell \) is the mean module execution time. The number of modules to be allocated is varied from 4 to 50. The execution time of a module is exponentially distributed with mean 1.0 unit of time. The IMC volume between two communicating modules is uniformly distributed over \((0, 10)\) data units. The worst-case recovery time \( t_{rec} \) is exponentially distributed with mean 1.0 unit of time. \( P_{ND} \) is assumed to be \( 1 - 10^{-5} \). The precedence constraints and the timing requirements of the TG are also randomly generated.

Before running experiments, we eliminated the TGs which were definitely infeasible. Infeasibility is detected by calculating release times and latest completion times of all modules, while ignoring all IPC times. If the interval between the latest completion time and the release time is less than the execution time for some module(s) in all the component graphs of a TG, this TG is infeasible, and is not considered any further. All experiments were performed on a SPARC station running the SUNOS 4.1.2 operating system.

The proposed scheme strikes a balance between the fault-tolerance achieved by replicating modules and the system capacity available for the timely completion of all tasks in the TG. Consider the example of replicating and allocating the TG in Fig. 1 (b) to a distributed system represented by a complete graph of 3 PNs. The worst-case recovery time \( t_{rec} \) is 1.2 units of time. The modules that should be replicated are those belonging to \( T_2 \) and \( T_3 \), since the execution path \( M_2 \rightarrow M_3 \rightarrow M_4 \rightarrow M_5 \rightarrow M_{10} \rightarrow M_{11} \) is critical subject to \( T_3 \)'s deadline and cannot tolerate any recovery delay. The same execution path cannot tolerate any IPC delay either, and hence, the MA scheme allocates all the modules that lie on this critical path to the same PN. Moreover, the system can accommodate up to 2 replicas of each of the modules on the critical path while ensuring the timely completion of all tasks. That is, the best degree of module replication is 2, and the best allocation is to assign modules \( T_1 \) to \( N_1 \), modules \( T_2 \cup T_3 \) to \( N_2 \), and the replicated modules of \( T_2 \cup T_3 \) to \( N_3 \).

Another interesting finding is that heavily communicating modules may not necessarily be allocated to the same PN. For example, consider replicating and allocating the TG in Fig. 4(a) to a distributed system of 4 PNs. The attributes of the TG are specified in the figure. The only critical path is \( M_2 \rightarrow M_3 \), and thus \( M_2 \) and \( M_3 \) are replicated. As shown in Fig. 4(b), the best degree of module replication is 2, and the MA scheme allocates \( M_1, M_6 \) and \( M_7 \) to \( N_1; M_4, M_5, M_8 \) and \( M_9 \) to \( N_1; M_10 \) to \( N_2; M_3, M_4 \) and \( M_5 \) to \( N_3 \), and the replicas of \( M_2 \) and \( M_3 \) to \( N_4 \) so that all modules meet their latest completion times. Although the IMC between \( M_4 \) and \( M_5 \) is twice more than the others, \( M_4 \) and \( M_5 \) are allocated to different PNs. This is mainly because \( T_2 \) has a less tight timing constraint than others and can thus allow IPCs among its modules. This observation is in sharp contrast to the common notion that heavily communicating modules should always be co-allocated [18].

By virtue of the BB method, the MA scheme always yields the best allocation given both the original and replica modules. Moreover, as reported in [17], the MA scheme finds it at tractable computation costs for task systems with less than 50 modules and/or distributed systems with less than 40 PNs, and usually no more than 9% of the search tree vertices were visited before finding the best allocation for \( N \geq 6 \) and \( K \geq 3 \). This suggests that both the dominance relation
7 Conclusion

We have addressed the problem of replicating and allocating periodic task modules in a distributed real-time system subject to precedence and timing constraints, and intermodule communications. The probability of no dynamic failure is used as the objective function to ensure all real-time tasks to be completed by their deadlines. The modules that have stringent timing constraints and cannot tolerate a worst-case recovery delay are selected for replication using the critical path analysis. The optimal number of replicas of each selected module (with respect to a pre-determined \( P_{ND} \)) and the assignment/scheduling of both original and replica modules are then determined by the MA scheme. The MA scheme not only assigns modules to PNs, but also uses the MS scheme to schedule all modules assigned to each PN.

An interesting finding from our numerical simulations is that sequentially-executing modules subject to the same timing constraints are usually chosen to be replicated. Moreover, these modules also tend to be allocated to the same PN by the MA scheme. Also, the common notion in general-purpose distributed systems that heavily communicating modules should be colocated [18] may not always be applicable to real-time systems. Only in case when there are enough resources to meet the timing requirements in the TG, the MA scheme assigns modules to minimize IPCs.

References