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 (P_{ND}) — 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 P_{ND}, 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 based on some application-dependent criterion and the system architecture under consideration, while the dynamic distribution of aperiodic 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 [5, 6, 7]; **(O2)** load balancing by minimizing the statistical variance of processor utilization [8, 9] or by maximizing the total rewards in the semi-Markov process that models the computer system [10]; **(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, P_{ND}, 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

The work described in this paper was supported in part by the ONR under Grants N00014-J-92-1080 and N0004-94-1-0229.

0363-8928/94 $3.00 © 1994 IEEE

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.

Ramamritham [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 Ramamritham 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 $\{p_i : i = 1, 2, ..., N_T\}$, 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-

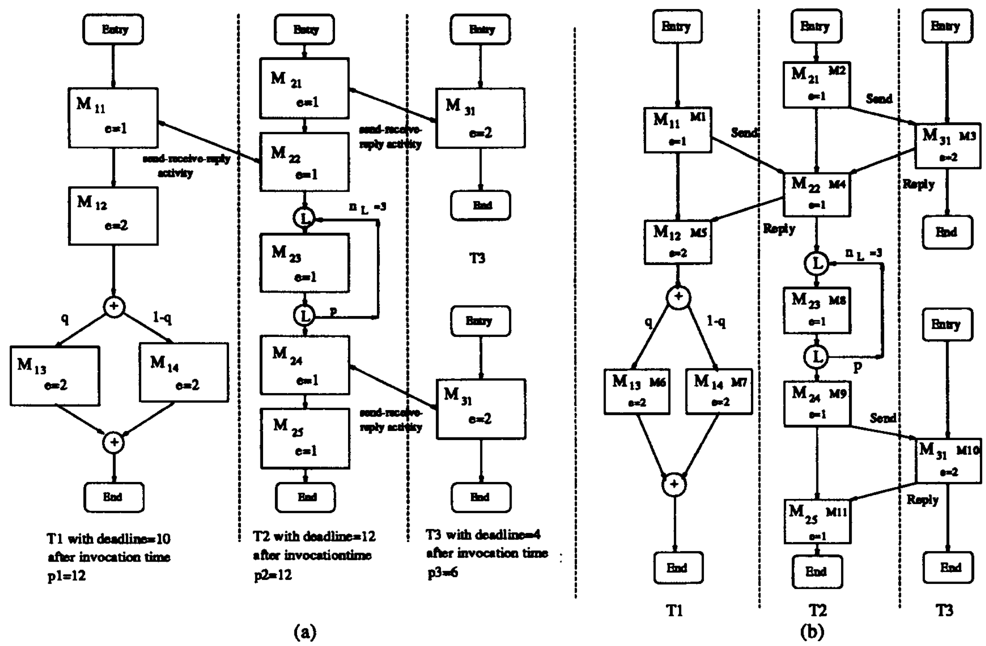

Figure 1: An example of task flow graph.

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_b 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

The distributed system considered here consists of K processing nodes (PNs). For ease of algorithm description, all PNs are assumed to have the same processing power and the same set of resources. The time required by an IMC within a PN is assumed to be negligible, while that between two PNs is expressed as the product of the IMC volume (measured in data units) and the link delay (measured in time units per data unit) between the two PNs on which the communicating modules reside.[1] The link delay could be the worst-case communication delay experienced by messages in the underlying time-constrained communication subsystem. Here we assume that the communication subsystem and the underlying protocol support time-constrained communications, and the worst-case delay experienced by messages is bounded and predictable. Two examples of such communication subsystems are the described in [19, 20]. No restriction is imposed on the topology of the communication subsystem.

3 Selection of Modules to Replicate

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

[1] The time for packetization and depacketization is lumped into module execution time.

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.,

$$LC_i - r_i < e_i + t_{rec}, \qquad (3.1)$$

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_L copies of its loop body, where n_L 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

$$LC_i = \min\{LC_i,\ \min_j\{LC_j - e_j : M_i \rightarrow M_j\}\}, \quad i = N-1, \ldots, 1, \qquad (3.2)$$

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

$$r_i = \max\{r_i,\ \max_j\{r_j + e_j : M_j \rightarrow M_i\}\}, \quad 2 \le i \le N, \qquad (3.3)$$

where r_1 is the invocation time of the task to which M_1 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

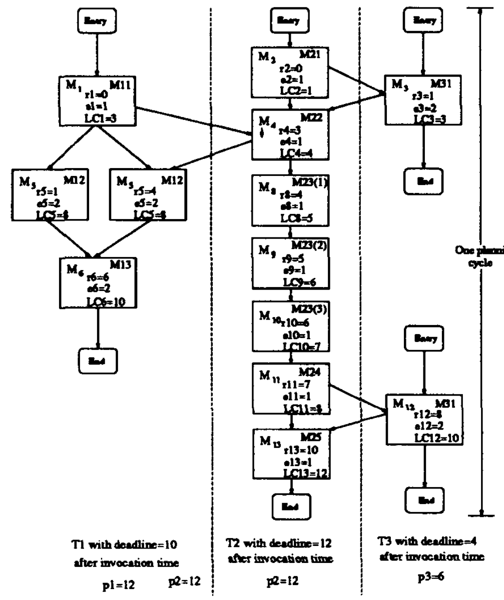

Figure 2: An example showing how r_i's and LC_i's are computed in the TG given in Fig. 1 (b).

r_4 of M_4 is calculated as

$$\begin{aligned} r_4 &= \max\{r_4, r_1 + e_1, r_2 + e_2, r_3 + e_3\} \\ &= \max\{0, 0+1, 0+1, 1+2\} = 3, \end{aligned}$$

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 $M_2 \rightarrow M_3 \rightarrow M_4 \rightarrow M_8 \rightarrow M_9 \rightarrow M_{10} \rightarrow M_{11} \rightarrow M_{12} \rightarrow M_{13}$ 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

$$\begin{aligned} \sum_{k=1}^{K} x_{ik} &= 1, \quad \text{for } i \in \text{modules not replicated, and} \\ \sum_{k=1}^{K} x_{ik} &= m_m, \quad \text{for } i \in \text{modules replicated,} \end{aligned}$$

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 $P_{ND}(x)$ is evaluated; for example, $P_{ND}(x) = 0$ if some of the tasks miss their deadline under x. The expression for $P_{ND}(x)$ 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[2] module release time and completion time (i.e., latest completion time − release time $\leq e_i + t_{rec}$), they will not be assigned to (and scheduled on) the same PN under an optimal allocation. This is because a module can be invoked only after its release time and must be completed by its latest completion time, if some of them were assigned to the same PN, they will not be able to complete in time, leading to $P_{ND} = 0$, since the interval between the release time and the latest completion time is bounded by $e_i + t_{rec}$. This ensures that module replicas are allocated to *distinct* PNs.

To determine m_m, we note that the larger m_m, the better fault-tolerance capability for the task system to deal with node failures. However, excessive replicas may jeopardize the timely completion of modules due to limited system resources. To determine m_m with respect to a pre-specified P^{req}_{ND}, we take the following steps:

Step 1. Initialize m_m to be 2.

Step 2. Augment the task flow graph with m_m replicas for each critical module selected for replication. Each module replica inherits the same execution time and precedence/timing constraints as its original. Let the augmented task graph be denoted as **TG**.

Step 3. Use the **MA** scheme to determine the assignment and scheduling of all modules in **TG**, and moreover, the objective value, P^*_{ND}, achieved.

Step 4. If $P^*_{ND} > P^{req}_{ND}$, then $m_m \leftarrow m_m+1$, and go to **Step 2**. If $P^*_{ND} = P^{req}_{ND}$, then stop and m_m is the number of replicas for each selected module. Otherwise, stop and $m_m \leftarrow m_m - 1$ is the number of replicas that should be used.

MA scheme: The **MA** scheme uses the **BB** method to enumerate all possible solutions by 'growing' the corresponding search tree. Each intermediate (leaf) vertex in the search tree corresponds to a partial (complete) allocation. The **BB** method is composed of two procedures: *branching* and *bounding*. The branching process generates the child vertices of an intermediate vertex x in the search tree, until an optimal solution is completely specified. Usually a dominance relation is derived to limit the number of child vertices generated from each intermediate vertex x without eliminating any path to an optimal solution. On the other hand, the bounding process calculates a tight upper bound of the objective function (UBOF) for each vertex x based on which one can decide whether or not x may lead to an optimal solution. If the UBOF of a vertex x is less than the current best objective value found in the search process, then x will never lead to an optimal solution, and should thus be pruned.

The **MA** scheme works as follows. All modules in the augmented task graph, **TG**, are numbered in acyclic order. The scheme begins with a null allocation x_0 which corresponds to the root of the search tree, and allocates modules in the order of their acyclic numbering. Let $\mathbf{TG}(x)$ denote the set of modules which are already allocated under x,[3] 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 $x \in AN$ corresponds to allocating the module, M_i, with the smallest acyclic number in $\mathbf{TG} \setminus \mathbf{TG}(x)$ to a PN, where $\setminus$ 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_k, during the interval $[r_i, LC_i]$ is smaller than e_i, and (2) the module, say M_j, scheduled to be executed last on N_k in $[r_i, LC_i]$ under a partial allocation x has tighter timing constraints than M_i (so no preemption on N_k to ensure the completion of M_i before LC_i), then allocating M_i to N_k is likely to miss M_i's latest completion time. Thus, N_k 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 x by allocating M_i to one of the candidate PNs. The UBOF, $\hat{P}_{ND}(y)$, of each newly-generated (intermediate) vertex y is calculated by scheduling modules $\in \mathbf{TG}(y)$ 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 $\hat{P}_{ND}(y)$ 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, x_0, of the search tree, which corresponds to a null allocation. Set $AN := \{x_0\}$.

Step 2. Set $\mathbf{TG}(x_0) := \emptyset$, $x_{opt} := x_0$, and the objective function value achieved by x_{opt}, $P^*_{ND} = 0.0 = \hat{P}_{ND}(x_0)$.

Step 3. While $AN \neq \emptyset$ **do**

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

[2] without considering possible IPC communication time between two modules.

[3] $\mathbf{TG}(x) = \mathbf{TG}$ if x is a complete allocation.

Step 3.1. Node Selection Rule:

Step 3.1.1. Select the vertex $x \in AN$ with the largest $\hat{P}_{ND}(x)$.

Step 3.1.2. If $\hat{P}_{ND}(x) < P^*_{ND}$, terminate **MA**, and x_{opt} is the optimal solution. Otherwise, set M_i to be the module $\in \mathbf{TG} \setminus \mathbf{TG}(x)$ 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 x 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 $\mathbf{TG}(y)$ under y and calculate the UBOF, $\hat{P}_{ND}(y)$.

Step 3.3.2. If $\hat{P}_{ND}(y) \leq P^*_{ND}$, 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, $\hat{P}_{ND}(y)$ is the actual P_{ND} achieved under y. Since $\hat{P}_{ND}(y) > P^*_{ND}$, set $x_{opt} := y$ and $P^*_{ND} = \hat{P}_{ND}(y)$ 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 $P(T_\ell$ is timely completed under $x)$. $P_{ND}(x)$ can then be calculated from $P(T_\ell$ is timely completed under $x)$, $\forall T_\ell$.

5.1 Module Scheduling Scheme

To facilitate the description and analysis of MS, we introduce the following notations:

- $\mathbf{TG}_c$: a component task graph of $\mathbf{TG}$. If $\mathbf{TG}$ 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). For the time-being, we only need to know that $\mathbf{TG}_c$ contains neither loops nor OR-subgraphs.
- $\mathbf{TG}_c(x)$: the set of modules $\in \mathbf{TG}_c$ under x.
- $S_k(x) = \{M_i : x_{ik} = 1\}$: the set of modules assigned to N_k under x.
- C_i: the completion time of M_i.
- $f_i(C_i)$: the cost incurred by completing M_i at C_i.
- $\hat{e}_i$: the *modified* execution time of M_i, where

$$\hat{e}_i = \begin{cases} e_i & \text{if } M_i \text{ is scheduled upon } r_i \\ C_i - r_i & \text{otherwise.} \end{cases}$$

  $\hat{e}_i$ is used to include the effect of queueing M_i on the release times of modules that succeed M_i.
- $com_{ij}(x)$: 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 ℓ_{mn}.
- $n(k,\ell)$: the number of edge–disjoint paths between N_k and N_ℓ.
- $I(m,n,k,\ell)$: the indicator variable such that $I(m,n,k,\ell) = 1$ if ℓ_{mn} lies on one of the $n(k,\ell)$ edge disjoint paths between N_k and N_ℓ.
- $Y_{k\ell} = \frac{1}{n(k,\ell)} \sum_{m=1}^{K} \sum_{n=1}^{K} I(m,n,k,\ell) \cdot t_{mn}$: the delay (in time units per data unit) bet. N_k and N_ℓ.
- 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 $e(B) = \sum_{M_i \in B} e_i$.
- dg_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 $f_{max}(S_k(x)) \triangleq \max_{M_i \in S_k(x)} f_i(C_i)$ 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 $\mathbf{TG}_c$.

To obtain LC_i, of $M_i \in \mathbf{TG}_c$, let LC_i be initially set to the deadline of the task to which M_i belongs, and then modify LC_i as

$$LC_i = \min\{LC_i, \min_j\{LC_j - e_j - com_{ij}(x) : M_i \rightarrow M_j\}\}, \quad i = N_c - 1, ..., 1, \tag{5.2}$$

where the modules are numbered in acyclic order, N_c is the number of modules in $\mathbf{TG}_c$, and

$$com_{ij}(x) = \begin{cases} 0, & \text{if } M_i \to N_k \text{ and } M_j \to N_k \text{ under } x, \\ d_{ij}Y_{k\ell}, & \text{if } M_i \to N_k \text{ and } M_j \to N_\ell \text{ under } x. \end{cases}$$

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 $M_i \in \mathbf{TG}_c(x)$, let r_i be initially set to the invocation time of the task to which M_i belongs, and then modify r_i as

$$r_i = \max\{r_i, \max_j\{r_j + \hat{e}_j + com_{ji}(x) : M_j \to M_i\}\}, \quad 2 \le i \le N_c, \qquad (5.3)$$

where r_1 is the invocation time of the task to which M_1 belongs, and $\hat{e}_j = \max\{C_j - r_j, e_j\}$ is the *modified* execution time which equals the sum of M_j's execution time, e_j, and M_j's queueing time (if M_j is not scheduled to be executed upon its release). $\hat{e}_j$ is used to include the effect of queueing 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 $\mathbf{TG}_c(y)$ 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 $\hat{e}_j$'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 $\in S_k(x)$ in the order of nondecreasing release times. We then decompose $S_k(x)$ into blocks, where a block $B \subset S_k(x)$ is defined as the minimal set of modules processed without any idle time from $r(B) = \min_{M_i \in B} r_i$ until $c(B) = r(B) + e(B)$, where $e(B) = \sum_{M_i \in B} e_i$. 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 dg_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 $\hat{B} \triangleq \{M_i : M_i \in B, dg_i = 0\}$, i.e., modules without successors in B, and then select a module M_m such that

$$f_m(c(B)) = \min_{M_i \in \hat{B}} f_i(c(B)), \qquad (5.4)$$

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

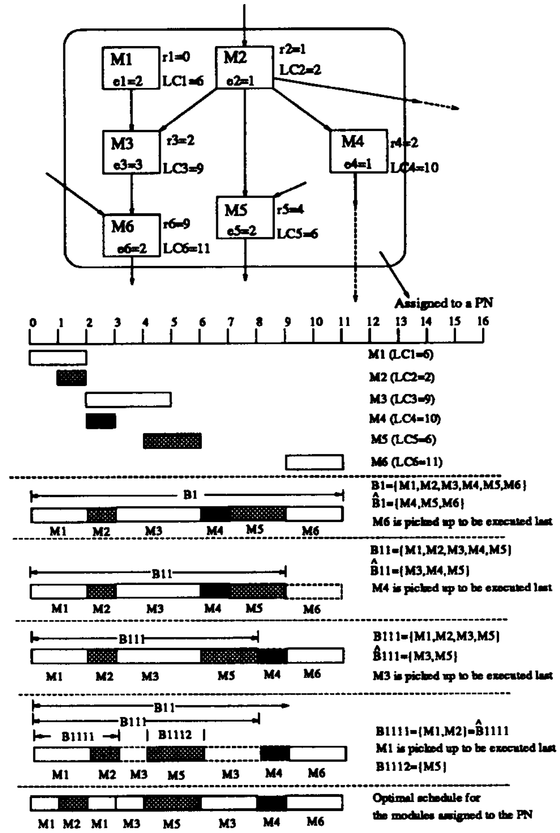

Figure 3: An example showing how the **MS** scheme schedules the modules assigned to a PN.

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 $\hat{B}_1, \hat{B}_2, ..., \hat{B}_{\hat{b}}$.

Sched2: A schedule for M_m, which is given by $[r(B), c(B)] - \cup_{j=1}^{\hat{b}}[r(\hat{B}_j), c(\hat{B}_j)]$, where $r(B) = \min_{M_i \in B} r_i$ and $c(B) = r(B) + e(B)$ with $e(B) = \sum_{M_i \in B} e_i$.

For this optimal schedule, we have

$$f^*_{max}(B) \text{ with the above restriction } = \max\{f_m(c(B)), f^*_{max}(B - \{M_m\})\} \le f^*_{max}(B) \quad (5.5)$$

where the last inequality comes from: (i) $f^*_{max}(B) \triangleq \min \max_{M_i \in B} f_i(C_i) \ge \min_{M_i \in B} f_i(c(B)) = \min_{M_i \in \hat{B}} f_i(c(B)) = f_m(c(B))$ by the way $\hat{B}$ was constructed from B and Eq. (5.4); (ii) Since $B - \{M_i\}$ is a subset of B, $f^*_{max}(B) \ge f^*_{max}(B - \{M_i\})$, $\forall M_i$.

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 $\hat{B}_1$, $\hat{B}_2$,..., $\hat{B}_b$, 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 one block: $B_1 = \{M_1, M_2, M_3, M_4, M_5, M_6\}$ from $[0, 11]$ (i.e., $r(B_1) = 0$, $e(B_1) = 11$, and $c(B_1) = 11$). Moreover, we have $\hat{B}_1 = \{M_4, M_5, M_6\}$ and select $M6$ to be processed only when no other modules are waiting since $LC_6 > LC_4 > LC_5$. $B_1 - \{M_6\}$ consists of one subblock: $B_{11} = \{M_1, M_2, M_3, M_4, M_5\}$ from $[0, 9]$. $\hat{B}_{11} = \{M_3, M_4, M_5\}$, and we select M_4 to be processed last since $LC_4 > LC_3 > LC_5$. $B_{11} - \{M_4\}$ consists of one subblock: $B_{111} = \{M_1, M_2, M_3, M_5\}$ from $[0, 8]$. $\hat{B}_{111} = \{M_3, M_5\}$, and we select M_3 to be processed last since $LC_3 > LC_5$. Now $B_{111} - \{M_3\}$ consists of two subblocks: $B_{1111} = \{M_1, M_2\}$ from $[0, 3]$ and $B_{1112} = \{M_5\}$ from $[4, 6]$. B_{1112} itself represents an optimal schedule, since B_{1112} consists of a single module. For B_{1111}, we have $\hat{B}_{1111} = \{M_1, M_2\}$ 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 for B_{1111} and B_{1112} (**Sched1**) and the schedule for M_3 (**Sched2**) which consists of $[0, 8] - [0, 3] \cup [4, 6]$. 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, $1 \leq i \leq N_c$, for $\mathbf{TG}_c$. This computation requires $O(N_c^2)$ time.

Step 2: Compute the release time r_i for $M_i \in \mathbf{TG}_c(x)$ 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,..., B_b 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, $1 \leq i \leq b$, update the outdegree, dg_j, of every $M_j \in B_i$. This update requires $O(|S_k(x)|^2)$ time for all B_i's $\subset S_k(x)$.

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 dg_j 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 $P_{ND}(x)$

We are now in a position to calculate $P(T_\ell$ 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 $S_k(x)$, $\forall k$. The completion time(s) of the last module(s) in $T_\ell \cap \mathbf{TG}$ under these schedules determines whether T_ℓ 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_ℓ to execute last if the last component in T_ℓ 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**. We first calculate the latest completion time, LC_i, of $M_i \in \mathbf{TG}$ 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_i's calculated is the worst-case latest completion time.

Second, we represent each loop $L_a \in \mathbf{TG}$ with the cascaded m copies of its loop body with probability $(1 - q_a)q_a^{m-1}$, 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_i calculated above, while the $(n_{L_a} - j)$-th copy of M_i bears the latest completion time $LC_i - j \cdot e(L_a)$, where $e(L_a)$ is the execution time of the loop body. Also, we represent each OR-subgraph $O_b \in \mathbf{TG}$ with its n-th branch with probability $q_{b,n}$, where $1 \leq n \leq n_{O_b}$, $q_{b,n}$ is the branching probability of the n-th branch of O_b, and n_{O_b} is the number of branches in O_b.

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_b in **TG**, then there are a total of $n_{L_a} \times n_{O_b}$ component graphs of **TG**, and with probability $p_c = (1 - q_a)q_a^{m-1} \cdot q_{b,n}$, the **TG** is represented by the **TG** with L_a replaced by the cascaded m copies of its loop body and O_b 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, $\mathbf{TG}_c$, of **TG**, we then calculate the release time, r_i, of $M_i \in \mathbf{TG}_c$ 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 $\mathbf{TG}_c$. Note that in a component graph $\mathbf{TG}_c$, 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 $P_{ND}(x)$: We now calculate the probability $P(T_\ell$ is timely completed under $x)$, $\forall T_\ell \in \mathbf{TG}_c$. Let the critical time of $M_i \in T_\ell$, D_i, be defined as the latest time M_i should be completed for the timely completion of only the task T_ℓ. Note that D_i can be obtained in the same way as LC_i except that the precedence relations, $M_i \rightarrow M_j$ when $M_j \notin T_\ell$, are ignored. That is, let D_i be initially set to the deadline of T_ℓ to which M_i belongs. Then, D_i is modified as:

$$D_i = \min\{D_i, \min_{M_j \in T_\ell} \{D_j - e_j - com_{ij}(x) : M_i \rightarrow M_j\}\},$$

$$i = N_c - 1, N_c - 2, ..., 1. \quad (5.6)$$

Obviously, $D_i \geq LC_i$. Also, let

$$\hat{T}_\ell \triangleq \{M_i : M_i \in T_\ell \cap \mathbf{TG}_c, dg_i = 0 \text{ w.r.t. } T_\ell \cap \mathbf{TG}_c\} \quad (5.7)$$

be the set of modules without any successor in $T_\ell \cap \mathbf{TG}_c$. Then, the probability $P(T_\ell$ is timely completed under x in $\mathbf{TG}_c)$ can be expressed as

$$P(T_\ell \text{ is timely completed under } x \text{ in } \mathbf{TG}_c) = \prod_{M_i \in \hat{T}_\ell} \delta(D_i - C_i), \quad (5.8)$$

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

$$P(T_\ell \text{ is timely completed under } x) = \quad (5.9)$$
$$\sum_{\{\mathbf{TG}_c\}} p_c \cdot P(T_\ell \text{ is timely completed under } x \text{ in } \mathbf{TG}_c),$$

where p_c is the probability that $\mathbf{TG}$ is represented by $\mathbf{TG}_c$, and $\{\mathbf{TG}_c\}$ is the set of component graphs of $\mathbf{TG}$, and finally

$$P_{ND}(x) = \prod_{\ell=1}^{N_T} P(T_\ell \text{ is timely completed under } x). \quad (5.10)$$

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 ℓ_{mn} is exponentially distributed with mean $0.1\bar{e}$, where $\bar{e}$ 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}^{req} 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 $M_2 \rightarrow M_3 \rightarrow M_4 \rightarrow M_8 \rightarrow M_9 \rightarrow M_{10} \rightarrow M_{11}$ is critical

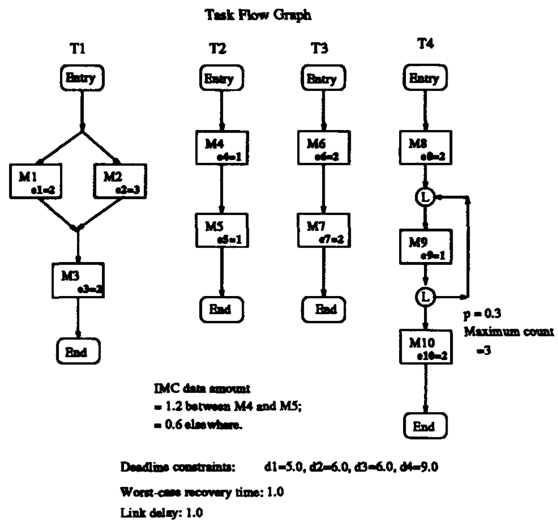

Figure 4: (a) The task graph and the system configuration used in Section 6.

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 $\in T_1$ to N_1, modules $\in T_2 \cup T_3$ 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_8, M_9 and M_{10} to N_2; M_2, M_3 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

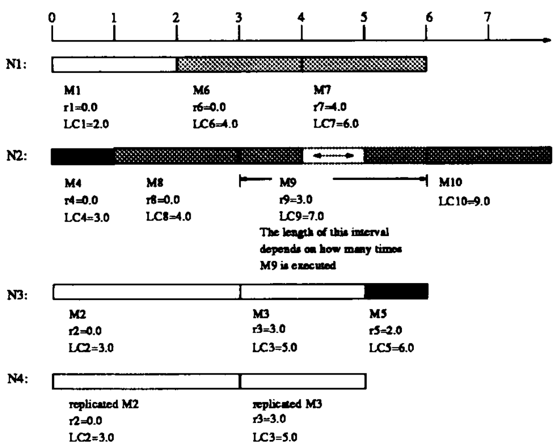

Figure 4: (b) Module replication, allocation and schedule for the configuration in Fig. 5 (a).

and the UBOF derived effectively prune unnecessary search paths at early stages of the **BB** process.

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}^{req}) 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 co-located [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

[1] D.-T. Peng and K. G. Shin, "Modeling of concurrent task execution in a distributed system for real-time control," *IEEE Trans. on Computers*, vol. C-36, pp. 500-516, Apr. 1987.

[2] K. G. Shin and Y.-C. Chang, "Load sharing in distributed real-time systems with state change broadcasts," *IEEE Trans. on Computers*, vol. C-38, pp. 1124-1142, Aug. 1989.

[3] K. Ramamritham, J. A. Stankovic, and W. Zhao, "Distributed scheduling of tasks with deadlines and resource requirements," *IEEE Trans. on Computers*, vol. C-38, pp. 1110-1123, Aug. 1989.

[4] K. G. Shin and C.-J. Hou, "Analytic models of adaptive load sharing schemes in distributed real-time systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, pp. 740-761, July 1993.

[5] P.-Y. Ma, E. Y. S. Lee, and M. Tsuchiya, "A task allocation model for distributed computing systems," *IEEE Trans. on Computers*, vol. C-31, pp. 41-47, Jan. 1982.

[6] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. on Software Eng.*, vol. SE-3, pp. 85-93, Jan. 1977.

[7] V. M. Lo, "Heuristic algorithms for task assignment in distributed systems," *IEEE Trans. on Computers*, vol. C-37, pp. 1384-1397, Nov. 1988.

[8] J. A. Bannister and K. S. Trivedi, "Task allocation in fault-tolerant distributed systems," *Acta Informatica*, vol. 20, pp. 261-281, 1983.

[9] A. N. Tantawi and D. Towsley, "Optimal static load balancing in distributed computer systems," *Journal of the ACM*, vol. 32, pp. 445-465, Apr. 1985.

[10] T. C. K. Chou and J. A. Abraham, "Load balancing in distributed systems," *IEEE Trans. on Software Engineering*, vol. SE-8, pp. 401-422, July 1982.

[11] C. C. Shen and W. H. Tsai, "A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion," *IEEE Trans. on Computers*, vol. C-34, pp. 197-203, Mar. 1985.

[12] W. W. Chu and L. M. T. Lan, "Task allocation and precedence relations for distributed real-time systems," *IEEE Trans. on Computers*, vol. 36, pp. 667-679, June 1987.

[13] W. W. Chu and K. K. Leung, "Module replication and assignment for real-time distributed processing systems," *Proceedings of the IEEE*, vol. 75, pp. 547-562, May 1987.

[14] D.-T. Peng and K. G. Shin, "Assignment and scheduling of communicating periodic tasks in distributed real-time systems," *Proc. of 9th Int'l Conf. on Distributed Computing Systems*, pp. 190-198, June 1989.

[15] S. M. Shatz and J.-P. Wang, "Model and algorithms for reliability-oriented task-allocation in redundant distributed-computer systems," *IEEE Trans. on Reliability*, vol. 38, pp. 16-27, Apr. 1989.

[16] K. G. Shin, C. M. Krishna, and Y. H. Lee, "A unified method for evaluating real-time computer controllers its application," *IEEE Trans. on Automatic Control*, vol. AC-30, pp. 357-366, Apr. 1985.

[17] C.-J. Hou and K. G. Shin, "Module allocation with timing and precedence constraints in distributed real-time systems," *IEEE Proc. 13th Real-Time Systems Symposium*, pp. 146-155, Dec. 1992.

[18] K. Ramamritham, "Allocation and scheduling of complex periodic tasks," *IEEE Proc. of 10th Int'l Conf. on Distributed Computing Systems*, pp. 108-115, May 1990.

[19] Q. Zheng and K. G. Shin, "On the ability of establishing real-time channels in point-to-point packet switched network." *IEEE Transactions on Communications*, March 1994.

[20] J. K. Strosnider and T. E. Marchok, "Responsive, deterministic IEEE 802.5 token ring scheduling," *Journal of Real-Time Systems*, vol. 1, pp. 133-158, Sept. 1989.

[21] K. R. Baker, E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan, "Preemptive scheduling of a single machine to minimize maximum cost subject to release dates," *Operations Research*, pp. 381-386, March-April 1983.