Decentralized load sharing in distributed real-time systems

Chang, Yi-Chieh, Ph.D.

The University of Michigan, 1991

DECENTRALIZED LOAD SHARING IN DISTRIBUTED REAL–TIME SYSTEMS

by

Yi-Chieh Chang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical Engineering)
in The University of Michigan
1991

Doctoral Committee:

Professor Kang G. Shin, Chairperson
Associate Professor Richard Brown
Professor Edward Davidson
Assistant Professor C.V. Ravishankar
Associate Professor C.T. Shih

To Shinyir

ACKNOWLEDGEMENTS

I would like to express the most sincere thanks to my advisor, Professor Kang G. Shin, who introduced me to the field of real–time computing systems four years ago. Without his guidance and constant encourangement I would not have been able to complete this dissertation. I also like to thank the other committee members: Professors Edward Davidson, Richard Brown, C.V. Ravishankar, and C.T. Shih for their comments and suggestions which have enhanced the quality of this dissertation.

Many thanks go to my friends and officemates for their help and valuable discussions throughout my graduate study.

Finally, I thank my parents, parents-in-law, brothers, and sisters for their support and encouragement. Especially, I like to thank my wife, Shinyir, for her support and patience.

The work reported in this dissertation was supported in part by the Office of Naval Research under contract N00014–K–85–0122 and the NASA under grant NAG–1–296.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Overview

Real–time computing is becoming a major discipline in Computer Science and Engineering due mainly to its increasing number of applications, such as command and control systems, process control systems, flight control systems, nuclear reactors, and life-support systems. Real–time systems differ from the traditional computer systems in that their correctness depends not only on the logical results of computation, but also on the time at which the results are produced. Failure to complete a (real–time) task before its deadline could cause a disaster [1, 2, 3]. Due to their potential for high performance and fault-tolerance with the multiplicity of processors, distributed systems are natural candidates to implement real–time applications [4, 5, 6, 7, 8, 9, 10]. Thus, scheduling real–time tasks to guarantee the deadlines in distributed real–time systems has become an active area of research [11].

Real–time tasks can be periodic or aperiodic. Periodic tasks have regular arrival times and deadlines while aperiodic tasks have random arrival times and deadlines. Scheduling periodic tasks in a uni-processor to meet hard deadlines was studied by Liu and Layland [12]. In terms of CPU utilization and deadline guarantees, the rate–monotonic and earliest deadline scheduling algorithms are shown to be optimal in static and dynamic priority cases, respectively [12]. The problem of scheduling real–time tasks on multiprocessor/distributed systems is shown to be NP-hard [13, 14], thereby leading to the development of many heuristic approaches [15, 16, 17, 18, 19]. Although some of these approaches, such as those in [16, 19], considered the possible arrival of aperiodic tasks, they assumed/implied that periodic tasks form the major portion of the task system. Extension of the rate monotonic and earliest deadline scheduling algorithms to multiprocessor or distributed systems has been studied extensively [2, 14, 20, 21, 16, 22, 17].

Aperiodic tasks are invoked by external interrupts, such as an abnormal reading from a sensor, obstacles encountered in case of a robot controller, or detection of a transient overload. Since aperiodic tasks arrive randomly, it is in general impossible to guarantee the completion of aperiodic tasks before their deadlines. Keeping the probability of missing deadlines, called the *probability of dynamic failure*, P_{dyn} [1], below a certain required level while minimizing the ensuing overhead is therefore the only meaningful course to take for scheduling aperiodic tasks.

Scheduling aperiodic tasks in distributed real–time systems is more difficult than scheduling periodic tasks, due to their random arrivals [23, 24]. Moreover, if task arrivals are unevenly distributed over the nodes in a distributed real–time system, some nodes may become overloaded, and thus, unable to complete all their tasks in time, while other nodes are left underloaded. In such a case, even if the total processing power of the system is sufficient to complete all incoming tasks in time, some tasks arriving at overloaded nodes may not be completed in time. One way to alleviate this problem is load sharing (LS); some of those tasks arriving at overloaded nodes are transferred to underloaded nodes for execution [5, 25, 26, 27].

Scheduling both periodic and aperiodic tasks in distributed real–time systems is far less addressed in literature than considering periodic or aperiodic tasks alone due to its difficulty and has not been treated extensively until recently [28, 29, 30, 31]. Sprunt *et al.* proposed a scheme that can guarantee the deadlines of periodic tasks while minimizing the response time of aperiodic tasks by exploiting the unused (by periodic tasks) CPU cycles [30]. In their study, aperiodic tasks are assumed to have soft deadlines with the lowest priority. Ramamritham *et al.* proposed to combine local and global scheduling approaches in distributed systems for both periodic and aperiodic tasks [19]. Periodic tasks are assumed to be known *a priori* and can always be scheduled locally. On the other hand, an aperiodic task may arrive at a node at any time and will be scheduled locally on the node if its deadline can be met there; otherwise, the task will be transferred to a remote node, which is called *global scheduling* in [19]. If none of the remote nodes can guarantee the deadline of this aperiodic task, it will be rejected and may seriously affect the system performance. Hence, the main effort in [19] was to design a heuristic, global scheduling policy so as to reduce the number of rejected aperiodic tasks. Three algorithms — called *bidding*, *focused addressing*, and *flexible* algorithms — were used to select a remote node for each aperiodic task which cannot be guaranteed locally. The basic idea of these algorithms is to collect state information, such as the *surplus processing power*, from other nodes such that, when a node cannot guarantee an aperiodic task locally, it will attempt to locate a remote node which can guarantee the task. The simulated best acceptance ratio of aperiodic tasks achieved in [19] is reported to be 96% of the total arrived aperiodic tasks. However, the impact of the rejected tasks on the system performance was not analyzed in [19]. It is also worth noting that the algorithms in [19] are variations of the bidding algorithm originally proposed for load balancing in general-purpose distributed systems [32, 33, 34, 35], where the primary goal is to reduce the average task response time. The bidding algorithm is not efficient for real–time applications due to the long delay of the bidding process [25, 36].

In this dissertation, a new Load Sharing Method with State-Change Broadcasts (LSMSCB) is proposed to schedule aperiodic tasks in distributed systems. In this approach, whenever the state of a node changes from "underloaded" to "fully-loaded"[1] and vice versa,

[1]These terms will be defined formally in Chapter 2

the node broadcasts this change to a set of nodes, called a *buddy set*, in its physical proximity. An overloaded node can select, without probing other nodes, the first available node from its *preferred list*, an ordered set of nodes in its buddy set. Preferred lists are constructed so that the probability of more than one overloaded node "dumping" their loads on a single underloaded node may be made very small. Performance of the proposed LSMSCB is evaluated with both analytic modeling and simulation. The LSMSCB is modeled first by an embedded Markov chain to which numerical solutions are derived.

The problem of scheduling both periodic and aperiodic tasks on distributed systems is studied in a new scheduling algorithm, called the reservation–based (RB) scheduling algorithm. In this approach, periodic tasks are scheduled first according to the rate monotonic priority algorithm [12]. Aperiodic tasks are assumed to have the lowest priority and are scheduled by utilizing the remaining unused CPU cycles. The RB scheduling algorithm is implemented on distributed real–time systems by modifying the LSMSCB. The set of periodic tasks on each node is pre-assigned and scheduled, and the CPU utilization on each node is known upon arrival of each aperiodic task. An aperiodic task will be scheduled locally if the unused CPU time is larger than its required computation time, otherwise the task will be either transferred to another node that can guarantee its deadline, or rejected if none of the nodes can guarantee its deadline.

1.2 Outline of the Dissertation

In a real–time system, the probability of missing a deadline must be kept as low as possible because the loss of a task may lead to a disastrous circumstance [1, 37]. Scheduling aperiodic tasks in distributed real–time systems is a load sharing problem, since tasks arriving at overloaded nodes must be transferred to idle or underloaded nodes to meet their deadlines [5, 25, 26].

A new load sharing method with state-change broadcasts (LSMSCB) for distributed real–time systems is proposed in Chapter 2. The LSMSCB is modeled with an embedded Markov chain to which numerical solutions are obtained by a two-step approximation. The numerical solutions are then used to compute the distribution of queue length (QL) at each node and the probability of meeting deadlines. As mentioned earlier, keeping P_{dyn} below a given required level while minimizing the resultant overhead is the only course to take for scheduling aperiodic tasks.

Deriving a closed–form distribution of QL to characterize the behavior of a real–time system is the subject of Chapter 3. Based on this closed-form distribution, 'optimal' LS solutions can be determined either by minimizing the communication overhead — such as the frequency of collecting state information and the number of task transfers — incurred by the LSMSCB while keeping P_{dyn} below any required level, or by minimizing P_{dyn} while keeping communication cost below any given level.

Chapter 4 identifies and solves two important issues associated with the use of preferred lists and buddy sets. First, when the number of overloaded nodes is less than, or equal to, the number of underloaded nodes, no more than one overloaded node should be allowed to select the same underloaded node as a receiver. Simultaneous selection of an underloaded node by multiple overloaded nodes results from lack of coordination among them, and will henceforth be called the *coordination* problem. Second, to minimize the task transfer delay, the buddy set of a node is composed of those nodes in its physical proximity (e.g., those one or two hops away). The overloaded nodes in a buddy set should be able to transfer their tasks to the nodes in different buddy sets such that those tasks arriving at overloaded nodes within a *hot region* — a region where the number of overloaded nodes is greater than that of underloaded nodes — can be shared throughout the entire system, not just by these nodes in the same buddy set. Formation of hot region(s) with an excessive number of overflow tasks will henceforth be called the *congestion* problem.

Chapter 5 addresses and analyzes two fault–tolerance issues associated with the

LSMSCB in hypercube multicomputers: (i) ordering fault–free nodes as preferred receivers of overflow tasks and (ii) developing a LS mechanism to handle node failures. The occurrence of node failures will destroy the original structure of a preferred list if the failed nodes are simply dropped from the list. Three algorithms are proposed to modify the preferred list to retain its original features regardless of the number of faulty nodes in the system. Based on the modified preferred lists, a simple mechanism is proposed to avoid/minimize task loss. Each node is equipped with a backup queue of its most preferred node. Whenever a node fails, its most preferred node will process the unfinished tasks in the backup queue, i.e., the tasks in the failed node's queue. By using the proposed algorithms, the failed node will be replaced by a fault-free node such that the node, which was originally supposed to select the failed node as its most preferred node, will always be backed up by a fault-free node.

Chapter 6 considers the problem of scheduling both periodic and aperiodic tasks in distributed real–time systems. A reservation–based (RB) scheduling algorithm is proposed and studied to guarantee the deadlines of periodic tasks while minimizing the probability of missing the deadlines of aperiodic tasks. A fraction R of CPU time is reserved in each unit cycle without violating the deadlines of periodic tasks. A *unit cycle* is the greatest common divisor (GCD) of all task periods [38]. The value of R is found to greatly affect the probability of guaranteeing aperiodic tasks even when the average CPU utilization on a node is fixed. The relation between the reserved fraction of CPU time and the probability of guaranteeing aperiodic tasks is studied. The maximum value of R is derived such that the probability of guaranteeing aperiodic tasks can be maximized.

Chapter 7 concludes this dissertation with the summary of contributions and unsolved research issues.

CHAPTER 2

DEVELOPMENT AND EVALUATION OF A DECENTRALIZED LOAD SHARING METHOD

2.1 Introduction

LS in general–purpose distributed systems has been studied extensively by numerous researchers [39, 40, 41, 42, 43, 44, 45]. Decisions on how to share loads among the nodes are either *static* or *dynamic* [46, 42, 47]. A static decision is independent of the current system state, whereas a dynamic decision depends on the system state at the time of decision. Static LS can also be viewed as nondeterministic allocation of tasks in a system [44, 48, 49, 50, 41], where an overloaded node N_i will transfer some of its tasks to node N_j with probability P_{ij}, which is independent of the current system state. Although static LS is simple and easy to analyze with queueing models, its potential benefit is limited since it does not adapt itself to the time-varying system state [40, 34, 33, 51, 52, 53, 54, 55]. For example, even when N_i is overloaded, it still has to accept tasks from other nodes with the same probability as if it were underloaded. On the other hand, when dynamic LS is used, an overloaded node can transfer its task(s) to other node(s) using the information on the current system state [40, 42, 43, 46, 56, 57]. Since any dynamic policy requires each node to know states of the other nodes, it is inherently more complex than any static policy. The advantage of a dynamic policy is that it adapts itself to the time-varying system state, and thus, can ease the difficulty associated with static LS.

LS algorithms can be source–initiated or server–initiated, depending on which node initiates task transfer. The node at which external tasks arrive is the source (sender) node, and the node that processes these tasks is the server (receiver) [39]. In the source–initiated approach, an overloaded source node initiates the transfer of a newly arriving external task based on some strategies, while in the server–initiated approach, an underloaded server will probe each of the potential source nodes to share its load with. LS algorithms are further divided into several levels according to the amount of information required for them. After analyzing and comparing the performance of these algorithms, Wang and Morris [39] concluded that an algorithm that collects more information will generally produce better results and that the server–initiated approach will usually outperform the source–initiated approach if the task transfer cost is not significant.

As was discussed by Eager *et al.* [40], LS is composed of a *transfer policy* and a *location policy*. The transfer policy determines when a node should transfer its task(s), i.e., when a node becomes overloaded. The location policy determines where to send task(s) to. Their objective was to minimize the average system response time by moving tasks from overloaded nodes to underloaded ones. A simple threshold was used in the transfer policy; that is, whenever the queue length of a node exceeds this threshold, it will attempt to transfer its incoming task to another node. Three different location policies were simulated and compared: random, threshold, and shortest. In the first method, an overloaded node (sender) will randomly choose another node (receiver) and transfer a task to that node. In the second method, the sender randomly selects one node and checks its load. If the load of this node is below the threshold, the sender will transfer a task to that node; otherwise another node is randomly chosen and checked. This process will repeat until a receiver is found or a prespecified checking limit is reached. If the sender cannot find a receiver, it will execute all its tasks. The shortest policy is a variation of the threshold policy in which the sender will choose a receiver with the shortest queue among the nodes it has probed. Their simulation results indicate that the random policy improves system performance significantly, as compared to the system without LS. The threshold policy can further improve system performance, as compared to the random policy. The improvement of the shortest policy is about the same as that of the threshold policy, although it requires more system state information than that of the threshold policy.

LS in distributed real–time systems is addressed far less in literature than that in general–purpose distributed systems. Kurose and Chipalkatti proposed a quasi-dynamic LS algorithm in a soft real-time system [5]. A job is considered *lost* if it is not completed before its deadline. Their primary objective was to reduce the probability of losing a job with LS. A node will transfer some of its jobs to another node if the unfinished workload exceeds its time constraints. The server was selected on a probabilistic basis which is independent of the current system state.

The location policy in most early work can be viewed as sender-initiated, since an overloaded node (sender) selects another node (a candidate receiver) and checks whether or not this node can share its load. If it can, the sender will transfer some of its tasks to that node; otherwise, the sender will probe another node. This process will repeat until a receiver is found or a prespecified limit is reached. There are two major drawbacks associated with this approach. First, the sender needs to probe other nodes before transferring any of its tasks. This will introduce an additional delay in completing the tasks to be transferred. Second, if only a few nodes in the system are underloaded, the sender may not be able to locate a receiver by probing only a limited number of nodes. In such a case, overloaded nodes must execute all their tasks locally, missing the deadlines of some of these tasks.

To alleviate the above drawbacks, we propose a load sharing method with state-change broadcasts (LSMSCB). In this method, each node needs to maintain the state information of only a small set of nodes, called a *buddy set*, in its physical proximity. Three thresholds, denoted by TH_u, TH_f and TH_v, are used to define the (loading) state of a node. A node is said to be *underloaded* if its queue length (QL) is less than or equal to TH_u, *medium-loaded* if $TH_u < QL \leq TH_f$, *fully-loaded* if $TH_f < QL \leq TH_v$, and *overloaded* if $QL > TH_v$. Whenever a node becomes fully-loaded (underloaded) due to the arrival and/or transfer (completion) of tasks, it will broadcast its change of state to all the other nodes in its buddy set. Every node that receives this information will update its state information by eliminating the fully-loaded node from, or adding the underloaded node to, its ordered list (called a *preferred list*) of available receivers. An overloaded node can select the first node in its preferred list and transfer a task to that node. Notice that our LS method is completely different from conventional receiver–initiated LS methods that are characterized in [39]. An underloaded server in the conventional receiver–initiated approach probes other nodes to share their work with. The difficulty of a receiver–initiated method lies in that once a node switches to H-state, this state–change must be multicast to other nodes so that this 'new' overloaded node may be drafted by an underloaded node as soon as possible. A large number of control messages need to be multicast if the load switches frequently between N-state and H-state. Although many protocols were considered in [35], this problem could not be solved completely without causing drafting delays to H-state nodes. By contrast, the LSMSCB is sender–initiated with receiver–initiated state updates, an overloaded node can locate, without any delay, an underloaded node using its preferred list. So, the state wiggling between TH_f and TH_v will not introduce any communication overhead at all, because the underloaded nodes need not know the exact state of overloaded nodes. Moreover, the state wiggling between TH_u and TH_f can be reduced by increasing the magnitude of $TH_f - TH_u$ without affecting the system performance as long as there are some underloaded nodes.

Section 2.2 discusses the problem of implementing the LSMSCB. Collection of state information and construction of preferred lists are detailed in Section 2.3. Section 2.4 presents one exact model, one approximate model, and an approximate solution to the exact model for the proposed LS method. In Section 2.5, the performance of the proposed LS method is evaluated with the models derived in Section 2.4 and is also simulated to verify the analytic results.

2.2 Problems Associated with the LSMSCB

In the LSMSCB, each node must maintain and update the state information of other nodes in its buddy set. An overloaded node can transfer a task to another node based on the state information without any probing delay. To implement this method, one must develop:

- Efficient means of collecting and updating state information; collection of the state information must not hamper normal communications, such as inter-task communications and task transfers.
- An automatic method for selecting a server node in case there are more than one underloaded node, and for minimizing the probability of more than one overloaded node simultaneously transferring their overflow tasks to the same underloaded node.

These issues are addressed below in detail.

2.2.1 State Information

To collect state information, one must decide from which nodes the information should be collected and how often this information should be updated. One straightforward method is for each node to collect and update state information from all other nodes in the system at a fixed time interval. However, it is very difficult to determine an appropriate collection and update interval which ensures the accuracy of the state information while keeping below an acceptable level the additional network traffic resulting from the collection of state information. Although a short interval (i.e., frequent state update) ensures the accuracy of the state information, this will introduce an $O(n^2)$ traffic overhead each time to collect the state information, where n is the number of nodes in the system. This may, in turn, severely delay normal inter-task communications and task transfers, thus degrading (rather than improving) system performance. On the other hand, the traffic overhead decreases as the frequency of state collection and update decreases. But, this may cause the state information recorded in a node to be obsolete. For example, if the state of a node has changed from underloaded to fully-loaded before the next update, other nodes may transfer their tasks to this already fully-loaded node based on the obsolete state information.

Ideally, each node should keep the state information as accurate and as up-to-date as possible while keeping the traffic overhead as low as possible. To achieve this goal, we propose *state–change broadcasts* within the buddy set of each node to collect and update the state information at the node. Thus, each node needs to maintain only the state information of a small set of nodes in the system, e.g., neighbors of the node. Each node will broadcast the change of state to all the other nodes in its buddy set only if it switched from underloaded to fully-loaded, and vice versa. Since a node will receive new information only when the state of a node in its buddy set has been changed, each node will have the exact state information of all the other nodes in its buddy set, as long as there is no significant delay in broadcasting state changes. Since the buddy set of each node is formed by those nodes in its physical proximity, the delay in broadcasting a state–change and/or transferring a task should not, in general, be too long. (See Section 2.5.1 for more on the broadcasting delay.) Moreover, as we shall see, the additional network traffic resulting from state–change broadcasts can be controlled by adjusting the two thresholds, TH_u and TH_f.

2.2.2 Preferred List

Under a sender-initiated location policy, an overloaded node will transfer a task to the underloaded node found first during the probing [39, 40, 5]. The problem in the proposed location policy is that an overloaded node may find more than one underloaded node in its buddy set and/or more than one overloaded node could select the same node to transfer their tasks to. One must therefore establish a rule for selecting a receiver among possible multiple underloaded nodes while minimizing the probability of more than one overloaded node simultaneously transferring tasks to the same underloaded node. A *preferred list* is proposed to counter this problem. Since each node maintains the state information of the nodes in its buddy set, one can order these nodes in each node's preferred list. The first node in this list is the *most preferred* and the second the *second most preferred*, and so on. Note that order of preference changes with time, e.g., if the most preferred node becomes overloaded, then the second most preferred node, if not overloaded, becomes the most preferred. An overloaded node will transfer its task(s) to its most preferred node available in the list.

Based on the system topology, the "static" order of nodes in each node's preferred list is so permuted that a node is the most preferred of one and only one other node in the corresponding buddy set. (This order does not change with time, although some of nodes will drop out of the list of available receivers when they become fully-loaded, and regain their spots when they become underloaded again.) Since each overloaded node is most likely to select the first node in its preferred list, the problem of more than one overloaded node "dumping" their loads on one node is unlikely to occur. Nevertheless, dumping could occur since, for example, the third most preferred node, say N_x, of an overloaded node N_o can become its most preferred while N_x is also the most preferred of another overloaded node. But the probability of this happening is small, since it will occur only after overloading all the nodes ahead of N_x in N_o's preferred list.

2.3 State Information and Preferred List

The nodes in the system are connected by an arbitrary network, and each node is equipped with a network processor which handles the usual communications and task transfers between nodes without burdening the node processor. A node has two sources of task arrivals, external tasks and transferred–in tasks, and one server (single node processor). The tasks arriving at each node may be executed locally, or remotely, at any other nodes in the system.

Every node is assumed to be *stable*, or its load density (i.e., the ratio of average external arrival rate to average service rate) is less than one. Thus, the need of load sharing arises when there are bursty arrivals of external tasks at one or more nodes in the system.

2.3.1 Collection of State Information

As mentioned earlier, three thresholds, TH_u, TH_f, and TH_v, are used to determine the state of a node. These thresholds can be QL or cumulative execution time (CET), depending on task characteristics. For example, if every task has the same (identically distributed) execution time, one can use the (average) QL to measure the workload of each node. However, if task execution times are neither identical nor identically distributed, one must use the CET to measure the workload of each node. By comparing each node's current workload with these thresholds, the node is determined to be in one of four possible states: under (U), medium (M), full (F), and over (V). QL will be used to measure a node's workload throughout this dissertation. A node is in U state if $QL \leq TH_u$, M state if $TH_u < QL \leq TH_f$, F state if $TH_f < QL \leq TH_v$, and V state if $QL > TH_v$.

A U-state node can accept one or more tasks from other nodes and complete them before their deadlines. A node in F state cannot accept tasks from any other nodes but can complete all of its own tasks in time. A node in V state cannot complete all of its own tasks in time, and thus, must transfer, if possible, some of its tasks to other node(s). Since a node in U state can share other nodes' work, it is said to be in *share mode.* A node in F state will neither accept tasks from other nodes nor transfer its own tasks to others, and is said to be in *independent mode.* A node in V state must transfer some of its tasks, and is said to be in *transfer mode.* Note that V is usually a transient state because, if arrival of a new task at a node switches the node to V state, the node will transfer this task to another underloaded node and then switch back to F state. However, if a V-state node cannot find a U-state node from its buddy set, it will be forced to remain in V state, missing some deadlines.

According to the state-change broadcasts, each node will broadcast the change of state to all the other nodes in its buddy set only when it switches from U to F and/or F to U. Upon receiving a state-change broadcast, every node in the corresponding buddy set will update its state information accordingly.

Two different thresholds, TH_u and TH_f, are used in the proposed LS method for the following reason. If only one threshold were used instead, a node would be in U (F) state when its QL is less (greater) than this threshold. In such a case a U-state node may switch to F state after receiving a task from another node, and an F-state node may switch back to U state after completing a task. Since a U-state node will accept tasks from other overloaded nodes, it is likely to switch to F state. On the other hand, an F-state node only accepts its own external tasks and is likely to switch back to U state, since every node is assumed to be stable. Thus, every node in the system will frequently switch between U and F, thereby increasing the network traffic to broadcast these state changes. Change of state occurs infrequently (frequently) when the difference between TH_u and TH_f is large (small).

The third threshold, TH_v, is used to avoid unnecessary task transfers. If we combine TH_f and TH_v into one threshold, then acceptance of one transferred task may make a node fully– as well as over– loaded. In this case, the fully– and over– loaded node must transfer its own newly arriving task to another node. Had it not accepted the transferred task, the node would not have to transfer the newly arriving task, and thus, one of the two task transfers would not have been needed. By introducing another threshold TH_v, each node will broadcast the change of state when it switches to F state, preventing other nodes from transferring tasks to that node. Since $TH_v - TH_f \neq 0$ and every node is assumed to be stable, a node is unlikely to become overloaded with its own arriving tasks. Thus, this difference can be used to control unnecessary task transfers.

The above three thresholds greatly influence system performance, such as the average task execution time, the probability of missing deadlines, and the traffic overhead of broadcasting state changes. These thresholds must therefore be determined to meet the system performance requirement. For example, in a real-time system, TH_v is a critical point below which a node processor can complete all queued tasks before their deadlines with a probability higher than required. The difference between TH_u and TH_f must be chosen to keep the traffic overhead induced by the state-change broadcasts below a specified value. These thresholds are also sensitive to system load and have to be adjusted as system load varies. (More on this will be discussed in Section 2.5.)

2.3.2 List of Preferred Nodes

As mentioned in Section 2.2, the purpose of constructing a preferred list for each node is to avoid the probing delay and the dumping problem. The cost of task transfer is an increasing function of the physical distance between the sender and receiver nodes. To reduce this cost, the receiver node should be located as closely to the source node as possible. The preferred list of each node is thus structured based on the number of hops between the source and receiver nodes. The first entry of a node's preferred list consists of its immediate neighbors, and the second entry consists of those nodes two hops away from the node, and so on. When there are more than one node in each entry, these nodes must be ordered to minimize the dumping problem.

To demonstrate how to order the nodes in each buddy set based on system topology, consider a regular[1] system with n nodes, N_1, N_2, $\ldots$, N_n, where the degree of N_i is k, $\forall i$. Link j of N_i is assigned a direction d_j, $0 \leq j \leq k-1$. N_i's "static" preferred list[2] is then constructed as follows. The set of N_i's immediate neighbors, denoted by P_1^i, is placed in the first entry of N_i's preferred list. The N_i's second entry, denoted by P_2^i, consists of the nodes in the first entry of every node in P_1^i, excluding the duplicated nodes. Generally, P_ℓ^i is the set of nodes which are listed in the first entry of every node in $P_{\ell-1}^i$, excluding the duplicated nodes.

Among the nodes in P_1^i, the node in direction d_0 is chosen to be the N_i's most preferred node in this entry, denoted by N_1^{i1}, and the node in direction d_1 is the N_i's second most preferred node in this entry, denoted by N_2^{i1}, and so on. The nodes in P_2^i are ordered as follows. The nodes in the N_1^{i1}'s first entry are checked according to their order in the entry. If a node in the N_1^{i1}'s first entry did not appear at any N_i's previous entry, it will be copied into the second entry of N_i in the same order as in the N_1^{i1}'s first entry. After all nodes in the first entry of N_1^{i1} are checked and copied, the nodes in the first entry of node N_2^{i1} will be checked and copied by the same procedure. This procedure will repeat until P_2^i is completed. The ordering of nodes in P_ℓ^i, $\forall \ell > 2$, can be determined similarly.

As an example, consider how the preferred list of each node in a 4-cube system (Fig. 2.1) is actually constructed. The identity (ID) of each node is coded with a 4-bit number, $b_3b_2b_1b_0$. The direction d_i of N_k is the link that connects N_k to a node whose ID differs from N_k's ID in bit position i, where $0 \leq i \leq 3$. One can now apply the above procedure to construct the preferred list for each node in the 4-cube system as shown in Fig. 2.2. Once each node's preferred list is constructed, an overloaded node N_i can select an

[1] A system is said to be *regular* if all node degrees are identical.

[2] This list is determined by the system topology and remains unchanged, but the availability of each node in this list changes with time.

underloaded node as follows. Check N_1^{i1} first; if it is underloaded, N_i will transfer a task to N_1^{i1}; otherwise, N_2^{i1} is checked, and so on. (This checking can easily be implemented with a pointer which is made to point to the first available node in the list.) If all the nodes in P_1^i are overloaded, N_i will sequentially check the nodes in P_2^i. If, albeit rare, an overloaded node cannot find any underloaded node from its preferred list, all of its tasks will be forced to execute locally.

The preferred list constructed above has the following advantages. First, since each node is the most preferred node of one and only one node in the "static" list, the probability of an underloaded node being selected by more than one overloaded node is very small. Second, the cost of task transfer is minimal, since a receiver node is selected, with a high probability, from the physical proximity of the source node. Moreover, the time overhead for selecting an underloaded node is negligibly small, because the time-consuming probing procedure used in most known methods [40, 5, 43] is not needed (More on the preferred lists will be addressed in Chapter 4).

Since the size of preferred list or buddy set will affect the probability of a task missing its deadline, it must be chosen to ensure that this probability is lower than the specified limit. However, a buddy set must not be too large because the larger the size of buddy set, the higher traffic overhead for the state–change broadcasts will result. Thus, there is a tradeoff between the capability of meeting deadlines and the traffic overhead caused by the state–change broadcasts. More on this will be discussed in Section 2.5.

2.4 Models for the LSMSCB

An embedded Markov chain is used to model the performance of the LSMSCB. We begin with the development of an exact model from which an approximate solution and an approximate model, called the *upper bound model*, will be derived. The exact solution will be shown to be (i) always upper bounded by the solution to the upper bound

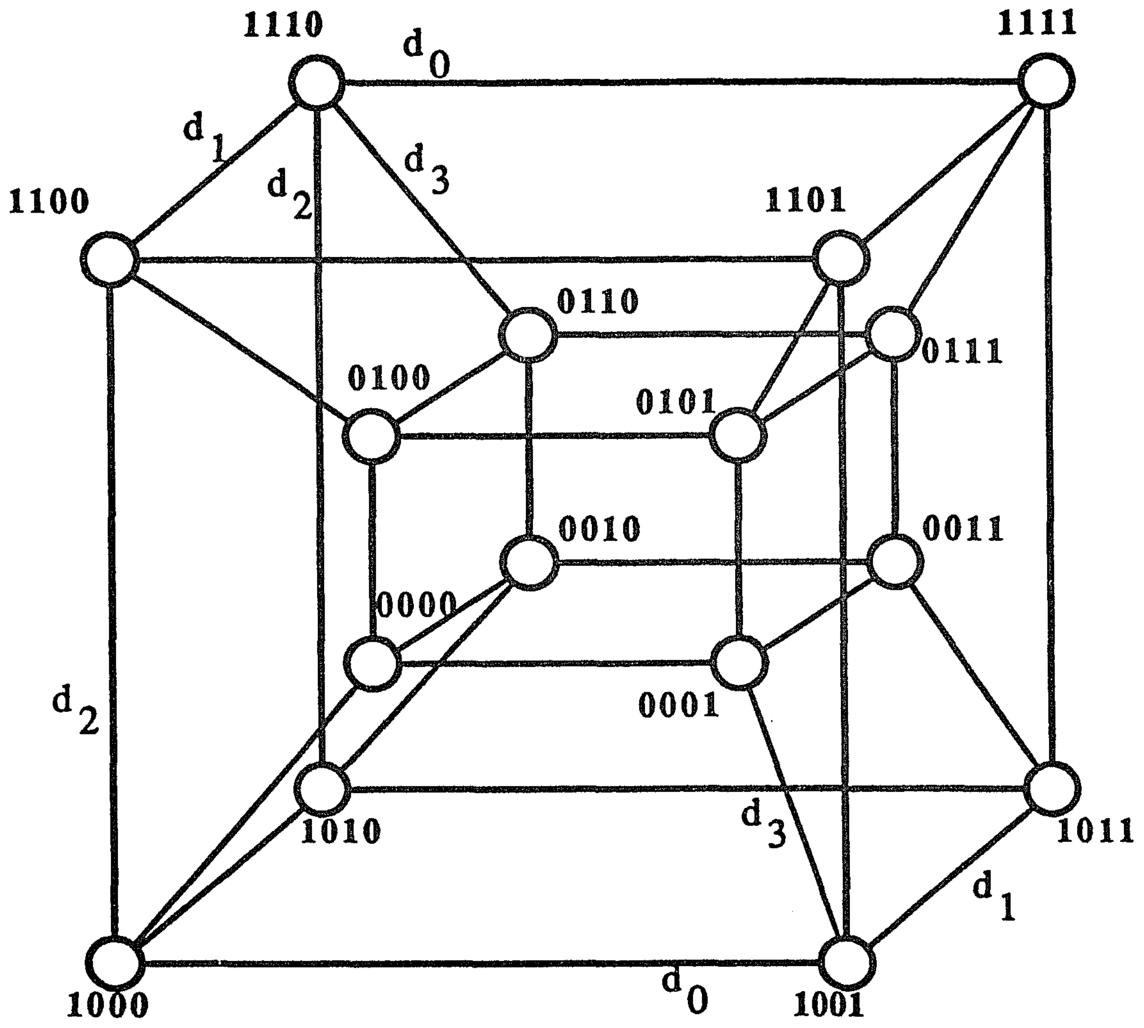


Figure 2.1: A 4-cube system.

Order of preference	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
node 0	1	2	4	8	6	10	12	3	5	9	14	13	11	7	15
node 1	0	3	5	9	7	11	13	2	4	8	15	12	10	6	14
node 2	3	0	6	10	4	8	14	1	7	11	12	15	9	5	13
node 3	2	1	7	11	5	9	15	0	6	10	13	14	8	4	12
node 4	5	6	0	12	2	14	8	7	1	13	10	9	15	3	11
node 5	4	7	1	13	3	15	9	6	0	12	11	8	14	2	10
node 6	7	4	2	14	0	12	10	5	3	15	8	11	13	1	9
node 7	6	5	3	15	1	13	11	4	2	14	9	10	12	0	8
node 8	9	10	12	0	14	2	4	11	13	1	6	5	3	15	7
node 9	8	11	13	1	15	3	5	10	12	0	7	4	2	14	6
node 10	11	8	14	2	12	0	6	9	15	3	4	7	1	13	5
node 11	10	9	15	3	13	1	7	8	14	2	5	6	0	12	4
node 12	13	14	8	4	10	6	0	15	9	5	2	1	7	11	3
node 13	12	15	9	5	11	7	1	14	8	4	3	0	6	10	2
node 14	15	12	10	6	8	4	2	13	11	7	0	3	5	9	1
node 15	14	13	11	7	9	5	3	12	10	6	1	2	4	8	0

Figure 2.2: Preferred lists of a 4-cube system.

model, called the *upper bound solution*, and (ii) very close to the approximate solution. Note that an embedded Markov chain is commonly used to analyze arbitrary task arrivals. Since (average) QL is used to measure workload, without loss of generality, one can assume (average) task execution time to be one unit of time. Let k_t and α_{k_t} be the number of task arrivals and the probability of having k_t arrivals during the interval $[t,\, t+1)$, respectively. For example, when the interarrival time of external tasks is exponentially distributed with rate λ, α_{k_t} can be calculated as shown in [58] by:

$$\alpha_{k_t} = \frac{\lambda^{k_t}}{k_t!} e^{-\lambda}. \tag{2.1}$$

Let x_t and x_{t+1} denote the QL at time t and $t+1$, respectively. Then,

$$x_{t+1} = \begin{cases} k_t & \text{if } x_t = 0 \text{ and } k_t \le TH_v \\ x_t + k_t - 1 & \text{if } x_t > 0 \text{ and } x_t + k_t \le TH_v + 1 \\ TH_v & \text{if } x_t + k_t > TH_v. \end{cases} \tag{2.2}$$

The above relation represents the case of ideal LS, since overloaded nodes are assumed to always find underloaded nodes to transfer their overflow tasks to.

Using Eq. (2.2), one can derive the probability distribution of QL. Two modifications must be made to include the effects of transferring and accepting tasks among the nodes in a buddy set. The first modification is to adjust the task arrival rate to include transferred–in tasks when a node is in U state. As shown in Fig. 2.3, the total arrival rate becomes $\omega = \lambda + \tau$, where λ and τ are the arrival rates of external and transferred–in tasks, respectively. A node's state transition probability depends on ω when the node is in U state, and thus, α's must be recalculated accordingly. Let α^*'s represent the transition probability corresponding to ω, whereas α's represent that corresponding to λ only. The second modification is made to the maximum QL. Since a node will always transfer tasks to other nodes when $QL > TH_v$, the QL of a node with ideal LS is bounded by TH_v.

To illustrate these modifications, consider the threshold pattern "1 2 3" (i.e., $TH_u = 1, TH_f = 2, TH_v = 3$) as an example, and let $q_i = P(QL = i)$, $\forall\ i$. Then,

$$\begin{aligned} q_0 &= \alpha_0^* \, q_0 + \alpha_0^* \, q_1 \\ q_1 &= \alpha_1^* \, q_0 + \alpha_1^* \, q_1 + \alpha_0 \, q_2 \\ q_2 &= \alpha_2^* \, q_0 + \alpha_2^* \, q_1 + \alpha_1 \, q_2 + \alpha_0 \, q_3 \\ q_3 &= (1 - \alpha_0^* - \alpha_1^* - \alpha_2^*) \, q_0 + (1 - \alpha_0^* - \alpha_1^* - \alpha_2^*) \, q_1 \\ & \quad + (1 - \alpha_0 - \alpha_1) \, q_2 + (1 - \alpha_0) \, q_3 \\ q_k &= 0 \ \text{ for all } k > 3. \end{aligned} \tag{2.3}$$

Note that the assumption that a task takes one unit of time to complete is used in the above equation.

As mentioned earlier, Eq. (2.3) represents ideal LS, i.e., overloaded nodes can always locate underloaded nodes to which their overflow tasks are transferred. In reality however, an overloaded node may not always be able to find an underloaded node from its buddy set. An embedded Markov chain is developed below to handle this realistic case.

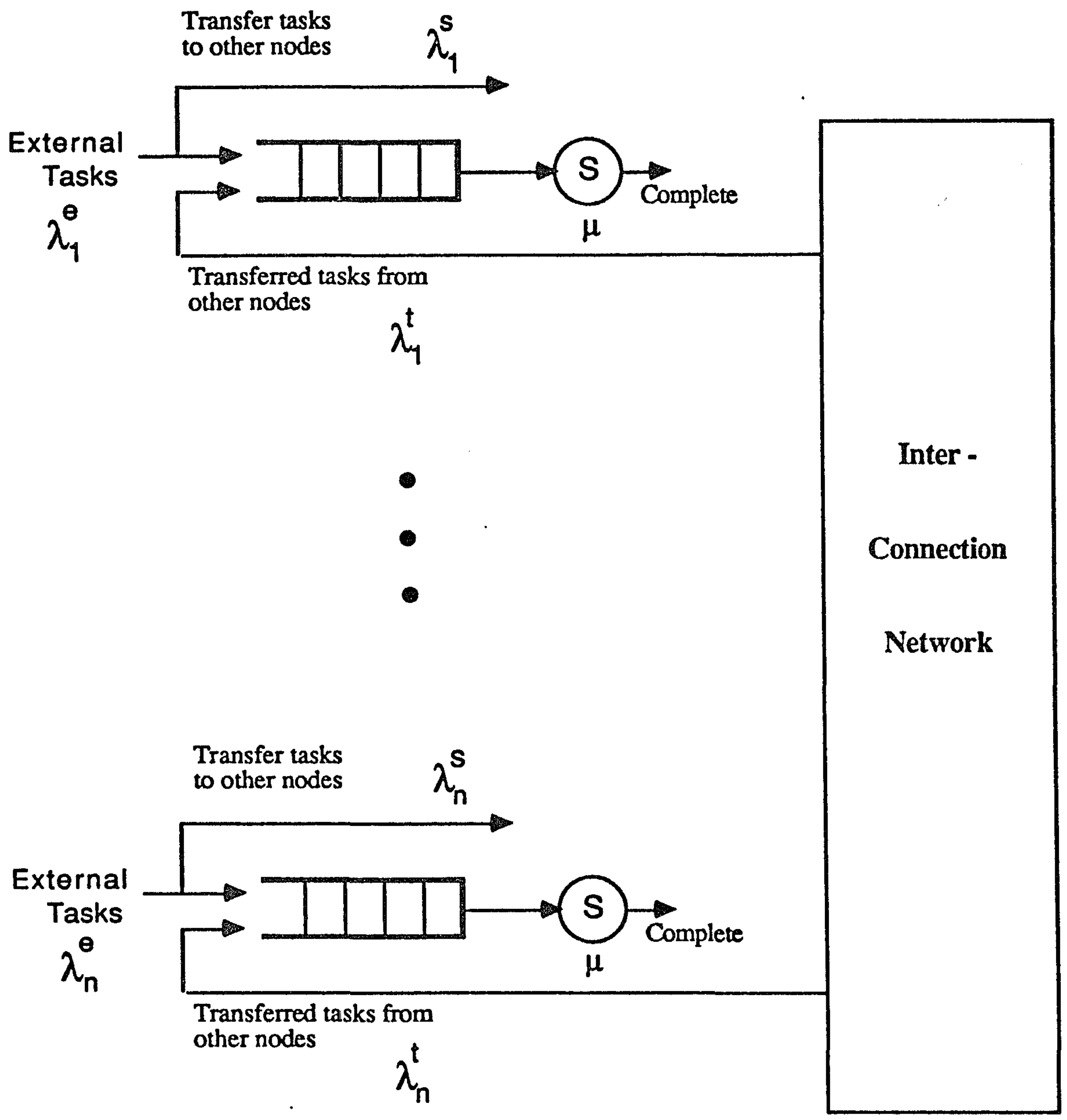

Transfer tasks to other nodes
λ_1^s
External Tasks
λ_1^e
S
μ
Complete
Transferred tasks from other nodes
λ_1^t
Inter - Connection Network
Transfer tasks to other nodes
λ_n^s
External Tasks
λ_n^e
S
μ
Complete
Transferred tasks from other nodes
λ_n^t

Figure 2.3: System model.

In the LSMSCB, the tasks in a node will be transferred to other nodes if its QL exceeds $TH_v + 1$ (TH_v) upon (before) completion of a task. A node can accept tasks from other nodes only when $QL < TH_f$. To transfer overflow tasks, the sharing capacity of each buddy set must be greater than or equal to the total number of overflow tasks in that buddy set. If this condition does not hold, an overloaded node's QL could grow larger than TH_v. To calculate the probability of a node's QL growing larger than TH_v, the following parameters are introduced. Let ε_i (θ_i) be the probability of having exactly (at least) i nodes available to share the overflow tasks within a buddy set. So, $\theta_i = 1 - \sum_{k=0}^{i-1} \varepsilon_k$ for $1 \leq i \leq n$, and $\theta_i = \varepsilon_i = 0$ for $i > n$, where n is the size of buddy set.

Assuming $x_t > 0$ for the previous example with threshold pattern "1 2 3", the number of overflow tasks in a node is $k_{ov} \equiv x_t + k_t - TH_v$. When $k_{ov} = 1$, $x_{t+1} = TH_v$ and the node will not transfer any task. When $k_{ov} = 2$, $x_{t+1} = TH_v$ if there is at least one node available for LS in its buddy set, and $x_{t+1} = TH_v + 1$ if none of the nodes in the buddy set is available for LS. Similarly, when $k_{ov} = \ell > 2$, $x_{t+1} = TH_v$ if there are at least $\ell - 1$ nodes available in its buddy set, or $x_{t+1} = TH_v + 1$ if there are exactly $\ell - 2$ nodes available, or, in general, $x_{t+1} = TH_v + j$ when there are exactly $\ell - (j + 1)$ nodes available. Then, the state transition relation can be rewritten as:

$$x_{t+1} = \begin{cases} k_t & \text{if } x_t = 0 \text{ and } k_t \leq TH_v \\ TH_v \text{ with prob. } \theta_{k_t - TH_v}, \text{ or } (TH_v + 1) \text{ with prob.} & \\ \quad \varepsilon_{k_t - TH_v - 1}, \ldots, \text{ or } k_t \text{ with prob. } \varepsilon_0 & \text{if } x_t = 0 \text{ and } k_t > TH_v \\ x_t + k_t - 1 & \text{if } x_t > 0 \text{ and} \\ & \quad x_t + k_t - 1 \leq TH_v \\ TH_v \text{ with prob. } \theta_{x_t + k_t - TH_v - 1}, \text{ or } (TH_v + 1) \text{ with prob.} & \\ \quad \varepsilon_{x_t + k_t - TH_v - 2}, \ldots, \text{ or } (x_t + k_t - 1) \text{ with prob. } \varepsilon_0 & \text{if } x_t > 0 \text{ and} \\ & \quad x_t + k_t - 1 > TH_v. \end{cases} \tag{2.4}$$

From the above relation, q_k's can be derived. For example, when the threshold pattern "1 2 3" is used, one can derive:

$$\begin{aligned} q_3 &= \left(\alpha_3^* + \sum_{i=1}^{\infty} \theta_i \, \alpha_{i+3}^*\right)(q_0 + q_1) + \sum_{i=2}^{4} \left(\alpha_{4-i} + \sum_{j=1}^{\infty} \theta_j \, \alpha_{j+4-i}\right) q_i \\ &\quad + \sum_{i=5}^{\infty} \left(\sum_{j=i-4}^{\infty} \theta_j \, \alpha_{j-i+4}\right) q_i \\ q_k &= \sum_{i=0}^{n} \varepsilon_i \, \alpha_{i+k}^* (q_0 + q_1) + \sum_{i=2}^{k+1} \left(\sum_{j=0}^{\infty} \varepsilon_j \, \alpha_{j+k-i+1}\right) q_i + \sum_{i=k+2}^{\infty} \left(\sum_{j=0}^{\infty} \varepsilon_{j+i-k-1} \, \alpha_j\right) q_i \\ &\quad \text{for } k = 4, \ldots \infty. \end{aligned} \tag{2.5}$$

Note that the q_k's for $k < TH_v = 3$ are the same as shown in Eq. (2.3), and q_k's for other threshold patterns can be derived similarly.

Although the above equations can be used to calculate the distribution of QL, ε_k's and θ_k's are in practice too complex to compute. For example, ε_k is the probability of having k nodes available for LS in an $n-$node buddy set, the calculation of which requires to consider $n!/(n-k)!\,k!$ different possibilities. The total number of possibilities that need to be considered for the calculation of ε_k for $k = 1, \ldots, n$ is 2^n. The analysis shows this number to be over 1,000 patterns when each buddy set contains 10 to 15 nodes. Furthermore, each of these patterns needs to be analyzed separately, since the probability of a node being in U state depends on the state of other nodes in the buddy set. Thus, it is extremely tedious to compute these parameters. To alleviate this difficulty, we develop the upper bound model and an approximate solution to Eq. (2.4). The former is used to derive ε_k's and a rough idea on the performance of the LSMSCB, while the latter to obtain approximate q_k's from Eq. (2.4) using the parameters derived from the upper bound model.

2.4.1 Upper Bound Model and Solution

Upper Bound Model

This model is derived under the assumption that every node can always transfer only one overflow task to another node and the rest of its overflow tasks are forced to queue at that node. Since on the average 50% of the computation capacity in each buddy set has to be available for LS,[3] the exact probability of an overloaded node being unable to transfer a task is always less than that derived from this model. The beauty of this model is the extreme simplicity in describing the state transition relation, as compared to the exact model. If $k_{ov} = 1$, then $x_{t+1} = TH_v$ and no task will be transferred. If $k_{ov} = 2$, then $x_{t+1} = TH_v$ with probability $\theta_1 = 1 - \varepsilon_0$, or $x_{t+1} = TH_v + 1$ with probability ε_0. When $k_{ov} = \ell$, $x_{t+1} = TH_v + \ell - 2$ with probability θ_1, or $x_{t+1} = TH_v + \ell - 1$ with probability ε_0. Summarizing the above leads to:

$$x_{t+1} = \begin{cases} k_t & \text{if } x_t = 0 \text{ and } k_t \le TH_v \\ (k_t - 1) \text{ with prob. } \theta_1, \quad \text{or} \quad k_t \text{ with prob. } \varepsilon_0 & \text{if } x_t = 0 \text{ and } k_t > TH_v \\ x_t + k_t - 1 & \text{if } x_t > 0 \text{ and } x_t + k_t - 1 \le TH_v \\ (x_t + k_t - 2) \text{ with prob. } \theta_1, & \\ \qquad \text{or} \quad (x_t + k_t - 1) \text{ with prob. } \varepsilon_0 & \text{if } x_t > 0 \text{ and } x_t + k_t - 1 > TH_v. \end{cases} \tag{2.6}$$

The distribution of QL can now be derived from this equation as follows:

$$\begin{aligned} q_0 &= \alpha_0^* q_0 + \alpha_0^* q_1 \\ q_1 &= \alpha_1^* q_0 + \alpha_1^* q_1 + \alpha_0 q_2 \\ q_2 &= \alpha_2^* q_0 + \alpha_2^* q_1 + \alpha_1 q_2 + \alpha_0 q_3 \\ q_3 &= [\alpha_3^* + (1-\varepsilon)\alpha_4^*] q_0 + [\alpha_3^* + (1-\varepsilon)\alpha_4^*] q_1 + [\alpha_2 + (1-\varepsilon)\alpha_3] q_2 \\ &\quad + [\alpha_1 + (1-\varepsilon)\alpha_2] q_3 + [\alpha_0 + (1-\varepsilon)\alpha_1] q_4 + (1-\varepsilon)\alpha_0 q_5 \\ q_4 &= [\varepsilon\alpha_4^* + (1-\varepsilon)\alpha_5^*](q_0 + q_1) + [\varepsilon\alpha_3 + (1-\varepsilon)\alpha_4] q_2 + [\varepsilon\alpha_2 + (1-\varepsilon)\alpha_3] q_3 \\ &\quad + [\varepsilon\alpha_1 + (1-\varepsilon)\alpha_2] q_4 + [\varepsilon\alpha_0 + (1-\varepsilon)\alpha_1] q_5 + (1-\varepsilon)\alpha_0 q_6 \\ q_k &= [\varepsilon\alpha_k^* + (1-\varepsilon)\alpha_{k+1}^*](q_0 + q_1) + \sum_{j=2}^{k+1} [\varepsilon\alpha_{k-j+1} + (1-\varepsilon)\alpha_{k-j+2}] q_j \\ &\quad + (1-\varepsilon)\alpha_0 q_{k+2} \quad \text{for all } k \ge 4, \end{aligned} \tag{2.7}$$

[3]Otherwise, load sharing is usually infeasible, and thus, should not be considered.

where $\varepsilon = \varepsilon_0$. Eq. (2.7) can be rewritten in vector form: $Q = AQ$, where $Q = [q_0, \ldots, q_n]^T$, A is an $n \times n$ coefficient matrix, and n is the size of buddy set. Using Eq. (2.7) and $\sum_{i=0}^{n} q_i = 1$, one can solve for Q, the upper bound solution.

The upper bound solution bounds the exact solution for the following reason. The only difference between the exact model Eq. (2.4) and the upper bound model Eq. (2.6) is that transitions to queue lengths $TH_v, TH_v + 1, \ldots, (x_t + k_t - 2)$ in Eq. (2.4) are combined into a single transition to $QL = x_t + k_t - 2$ in Eq. (2.6). Since $x_t + k_t - 2 \geq TH_v$, the transition to a $QL > TH_v$ is exaggerated in Eq. (2.6). Thus, the solution to the exact model will be bounded by the upper bound solution when $k > TH_v$. Note that the upper bound model is identical to the exact model when $k \leq TH_v$.

Solving Upper Bound Model

The upper bound model is analyzed first to get a rough idea on the performance of the LSMSCB. ω's and ε must be known before solving the upper bound model for q_k's. On the other hand, these parameters depend on q_k's, and thus, the model cannot be solved for q_k's without knowing ω's and ε. A two-step approximation approach is taken to handle the difficulty associated with this recursion problem. In the first step, the model is solved for τ and q_k's with $\varepsilon = 0$. The resulting q_k's are still an upper bound for the exact solution. The second step is to compute ε based on the q_k's obtained in the first step.

By setting $\varepsilon := 0$, q_k's for $k \geq 3$ in the upper bound model become:

$$\begin{aligned} q_3 &= (\alpha_3^* + \alpha_4^*)(q_0 + q_1) + (\alpha_2 + \alpha_3)\, q_2 + (\alpha_1 + \alpha_2)\, q_3 \\ &\quad + (\alpha_0 + \alpha_1)\, q_4 + \alpha_0\, q_5 \\ q_4 &= \alpha_5^*(q_0 + q_1) + \alpha_4\, q_2 + \alpha_3\, q_3 + \alpha_2\, q_4 + \alpha_1\, q_5 + \alpha_0\, q_6 \\ q_k &= \alpha_{k+1}^*(q_0 + q_1) + \sum_{j=2}^{k+1} \alpha_{k-j+2}\, q_j + \alpha_0\, q_{k+2} \quad \text{for all } k > 4. \end{aligned} \tag{2.8}$$

The above equation can be solved by using an iterative method. Initially, τ is set to 0. One can compute q_k's and then τ from

$$\beta \equiv \left[\sum_{k=4}^{\infty}(k-3)\,\alpha_k^*\right]\,(q_0+q_1)+\sum_{i=2}^{4}\left[\sum_{k=5-i}^{\infty}(k-4+i)\,\alpha_k\right]\,q_i+\sum_{i=5}^{\infty}\left[\sum_{k=0}^{\infty}k\,\alpha_k\right]\,q_i. \qquad (2.9)$$

Note that β is the rate of task transfer out of a node. If all nodes' external task arrival rates are identical, then $\tau = \beta$. Otherwise, τ must be calculated by Eq. (2.10). After calculating τ, ω is obtained by adding λ to τ, and then q_k's are recalculated with the new ω, which will, in turn, change τ. This procedure will repeat until q_k's and τ converge to fixed values. (The convergence will be proved later in Theorem 1.)

Lemma 2.1 $\dfrac{d\,q_k}{d\,\omega}$ *satisfies the following properties:*

1. $\dfrac{d\,q_0}{d\,\omega} < 0$
2. $\displaystyle\sum_{k=0}^{\infty}\frac{d\,q_k}{d\,\omega} = 0$
3. $\left|\dfrac{d\,q_k}{d\,\omega}\right| < 1, \quad \forall\, k.$

Proof: Since the probability of a system being idle (q_0) will decrease as the task arrival rate increases, the first property holds. The second property holds because the sum of all q_k's is equal to 1, and thus, the sum of the variations of all q_k's must be equal to 0. The last property can be proved by contradiction. Suppose $\left|\dfrac{d\,q_k}{d\,\omega}\right| \geq 1$. Then q_k may become negative or greater than 1 if the variation of ω exceeds 1, a possible event when a U-state node is surrounded by more than one V-state node. However, q_k can be neither negative nor greater than 1. Contradiction. □

Lemma 2.2 $0 \leq \dfrac{d\,\tau}{d\,\omega} < 1.$

Proof:

$$\begin{aligned}\frac{d\,\tau}{d\,\omega} &= \left[\sum_{k=4}^{\infty}(k-3)\,\frac{d\,\alpha_k^*}{d\,\omega}\right]\,(q_0+q_1)+\left[\sum_{k=4}^{\infty}(k-3)\,\alpha_k^*\right]\left(\frac{d\,q_0}{d\,\omega}+\frac{d\,q_1}{d\,\omega}\right)\\ &\quad+\sum_{i=2}^{4}\left[\sum_{k=5-i}^{\infty}(k-4+i)\,\alpha_k\right]\frac{d\,q_i}{d\,\omega}+\sum_{i=5}^{\infty}\left[\sum_{k=0}^{\infty}k\,\alpha_k\right]\frac{d\,q_k}{d\,\omega}\end{aligned}$$

$$= (1-\alpha_0^*-\alpha_1^*-\alpha_2^*)(q_0+q_1) + \left[\sum_{k=4}^{\infty}(k-3)\,\alpha_k^*\right]\left(\frac{d\,q_0}{d\,\omega}+\frac{d\,q_1}{d\,\omega}\right)$$
$$+\sum_{i=2}^{4}\left[\sum_{k=5-i}^{\infty}(k-4+i)\,\alpha_k\right]\frac{d\,q_i}{d\,\omega}+\sum_{i=5}^{\infty}\left[\sum_{k=0}^{\infty}k\,\alpha_k\right].$$

According to the definitions of α_k and α_k^*, each summation in the above equation is equal to, or less than, the average task arrival rate which is less than 1 in a stable system. (Recall that each node's service rate is assumed to be unity.) By the second and third properties of Lemma 1, the sum of the last three terms will be less than one. Furthermore, the first term will be much less than one, because the first three α^*'s usually dominate the determination of transition probabilities. Thus, the lemma follows. □

Theorem 2.1 *q_k's and τ derived from the above iterative method converge to fixed values in a finite number of steps.*

Proof: Let $d\,\omega^{(i)}$ and $d\,\tau^{(i)}$ be the variations of ω and τ at the i^{th} iteration, respectively. These parameters at the $(i+1)^{th}$ iteration are related to those at the i^{th} iteration by:

$$\begin{aligned} d\,\tau^{(i+1)} &= \frac{d\,\tau}{d\,\omega}\,d\,\omega^{(i)} \\ d\,\omega^{(i+1)} &= \tau^{(i+1)}-\tau^{(i)} = d\,\tau^{(i+1)}. \end{aligned}$$

Since $|\frac{d\,\tau}{d\,\omega}| < 1$ by Lemma 4, $d\,\tau$ at the $(i+1)^{th}$ iteration will be smaller than $d\,\omega$ at the i^{th} iteration. Since the variation of ω at the $(i+1)^{th}$ iteration is equal to that of τ at the $(i+1)^{th}$ iteration, we get $d\,\tau^{(i+1)} < d\,\tau^{(i)}$ and $d\,\omega^{(i+1)} < d\,\omega^{(i)}$. Thus, the variation of τ will decrease to zero after a finite number of iterations, and so is ω. Substituting the convergent τ and ω into Eq. (2.8), unique q_k's can be determined, i.e., q_k's also converge.

□

The numerical experiments show that τ and ω converge after only two to three iterations, indicating that the derivatives of τ and q_k with respect to ω are much smaller than 1.

Derivation of ε

The main difficulty in deriving ε_k's lies in the fact that the queue lengths in a buddy set depend on one another. Thus, the dependent LS environment is converted to an independent environment by using the Bayes theorem. To facilitate the description of this approach for an $(n+1)$–node buddy set, it is necessary to introduce the following variables.

- N^i_j: the j^{th} preferred node of N_i.
- x_i: the N_i's queue length.
- x_{ij}: N^i_j's queue length.
- x^i_{jk}: the queue length of the k^{th} preferred node of N^i_j.
- β_i: the rate of task transfer out of N_i.
- β_{ij}: the rate of task transfer out of N^i_j.
- γ_i: the rate of task transfer out of N_i given that N_i is not in sharing mode.
- γ_{ij}: the rate of task transfer out of N^i_j given that N^i_j is not in sharing mode.
- τ_i: the rate of task transfer into N_i.

It is easy to see that $\gamma_i > \beta_i$, since tasks are not actually transferred out of a node unless the node is in V state and β_i is the average transfer–out rate over the entire time period of interest.

Let N_0 be the node under consideration, then

$$\varepsilon = P(x_{01} \geq TH_f, \ldots, x_{0n} \geq TH_f) \qquad (2.10)$$

$$\begin{aligned}\tau_0 &= \beta_{01}\, P(x_0 \leq TH_u) + \beta_{02}\, P(x_0 \leq TH_u, x^0_{21} \geq TH_f) \\ &\quad +\beta_{03}\, P(x_0 \leq TH_u, x^0_{31} \geq TH_f, x^0_{32} \geq TH_f) + \cdots \\ &\quad +\beta_{0n}\, P(x_0 \leq TH_u, x^0_{n1} \geq TH_f, x^0_{n2} \geq TH_f, \ldots, x^0_{nn} \geq TH_f). \qquad (2.11)\end{aligned}$$

Note that τ_0 derived from Eq. (2.10) would be identical to that derived from Eq. (2.9) if all nodes have the same external task arrival rate. Using Eq. (2.9) in such a case, for

$j = 1, \ldots, n$

$$\gamma_{0j} = \sum_{i=2}^{4} \left[\sum_{k=5-i}^{\infty} (k-4+i)\, \alpha_k \right] \frac{q_i}{P^{nsh}} + \sum_{i=5}^{\infty} \left[\sum_{k=0}^{\infty} k\, \alpha_k \right] \frac{q_i}{P^{nsh}}, \tag{2.12}$$

where $P^{nsh} = 1 - q_0 - q_1$. Using the Bayes formula, the probability of both N_1^0 and N_2^0 not being in sharing mode can be calculated by

$$P(x_{01} \geq TH_f, x_{02} \geq TH_f) = P(x_{01} \geq TH_f)\, P(x_{02} \geq TH_f \mid x_{01} \geq TH_f). \tag{2.13}$$

Since the dependence between queue lengths is included in ω, its effect can be included by adjusting the rate of task transfer into N_2^0 given that N_1^0 is not in sharing mode. So, the conditional probability $P(x_{02} \geq TH_f \mid x_{01} \geq TH_f)$ can be equated to $P(x_{02} \geq TH_f)$, while the ω of N_2^0 must be adjusted to reflect the effect of N_1^0's unavailability. As shown in Eq. (2.12), such an adjustment will increase the rate of task transfer out of N_1^0 given that it is not in sharing mode, which will, in turn, increase N_2^0's task transfer–in rate. Moreover, N_0 will select N_2^0 as the most preferred node given that N_1^0 is not in sharing mode, and thus, the task transfer–in rate of N_2^0 should be recalculated. For notational convenience, let N_2 represent the node N_2^0 under consideration, then

$$\begin{aligned}\tau_2 &= \beta_{21}\, P(x_2 \leq TH_u) + \beta_{22}\, P(x_2 \leq TH_u) + \beta_{23}\, P(x_2 \leq TH_u, x_{31}^2 \geq TH_f, x_{32}^2 \geq TH_f) \\ &\quad + \cdots + \beta_{2n}\, P(x_2 \leq TH_u, x_{n1}^2 \geq TH_f, x_{n2}^2 \geq TH_f, \ldots, x_{nn}^2 \geq TH_f). \end{aligned} \tag{2.14}$$

The first two terms of Eq. (2.14) represent the transferred-in tasks from the N_2's most preferred node and N_0 $(= N_2^2)$. Since N_1^0 is unavailable, N_2 becomes the most preferred node of both N_1^2 and N_0. Clearly, N_2's ω will be larger than those of N_0 and N_1^0. Hence, it is likely to switch to no-sharing mode when N_1^0 is in no-sharing mode. Similarly, the probability of all N_1^0, N_2^0, and N_3^0 not being in sharing mode can be calculated as:

$$\begin{aligned} P(x_{01} \geq TH_f, x_{02} \geq TH_f, x_{03} \geq TH_f) &= P(x_{01} \geq TH_f, x_{02} \geq TH_f) \times \\ &\quad\; P(x_{03} \geq TH_f \mid x_{01} \geq TH_f, x_{02} \geq TH_f) \\ &= P(x_{03} \geq TH_f)\, P(x_{01} \geq TH_f, x_{02} \geq TH_f) \end{aligned}$$

$$= \quad P(x_{03} \geq TH_f)\, P(x_{01} \geq TH_f)\, P(x_{02} \geq TH_f).$$

The ω of N_2^0 and N_3^0 must be recalculated as described above. The correctness of Eq. (2.10) can be verified as follows. When all nodes in the system have the same distribution of QL and the same β, Eq. (2.10) can be simplified as:

$$\begin{aligned}
\tau_0 &= \beta_{01}\, P(x_0 \leq TH_u) + \beta_{02}\, P(x_0 \leq TH_u)\, P(x_{21}^0 \geq TH_f) + \cdots \\
&\quad + \beta_{0n}\, P(x_0 \leq TH_u)\, P(x_{n1}^0 \geq TH_f)\, P(x_{n2}^0 \geq TH_f) \cdots P(x_{nn}^0 \geq TH_f) \\
&= \beta\, P(x_0 \leq TH_u) \left[1 + P(x_{21}^0 \geq TH_f) + P(x_{31}^0 \geq TH_f)^2 + \cdots + P(x_{n1}^0 \geq TH_f)^n\right] \\
&= \beta\, P(x_0 \leq TH_u)\, \frac{1 - P(x_{21} \geq TH_f)^{n+1}}{1 - P(x_{21} \geq TH_f)} \simeq \beta\, P(x_0 \leq TH_u)\, \frac{1}{1 - P(x_{21} \geq TH_f)} \\
&= \beta\, P(x_0 \leq TH_u)\, \frac{1}{P(x_{21} \leq TH_u)} \simeq \beta.
\end{aligned}$$

Consider a 4-cube system as an example, in which, without loss of generality, N_0 can be viewed as the center node for the derivation of ε. From Fig. 2.2, N_0's preferred list is $N_1 N_2 N_4 N_8 N_6 N_{10} N_{12} N_3 N_5 N_9 N_{14} N_{13} N_{11} N_7$. Since the nodes near the end of the list are unlikely to be selected for LS, the adjusted task transfer–in rate of these four nodes can be approximated by adding $\beta_0\, P(x_0 \leq TH_u)$ to the τ of these nodes. Since increasing task transfer–in rate will change QL, the q_k's of these nodes need to be recalculated for

$$\begin{aligned}
P(x_1 \geq TH_f, x_2 \geq TH_f, x_4 \geq TH_f, x_8 \geq TH_f) &= P(x_1 \geq TH_f)\, P(x_2 \geq TH_f) \times \\
&\quad\; P(x_4 \geq TH_f)\, P(x_8 \geq TH_f) \\
&\equiv \varepsilon^{(4)}. \qquad (2.15)
\end{aligned}$$

Note that the states of these four nodes are different from that of N_0, because their task transfer–in rates are higher than that of N_0. Thus, these nodes are more likely to be in V-state than N_0. Similarly, one can calculate the adjusted task transfer–in rates for N_5 — N_{10}. As shown in Fig. 2.2, each of these nodes has two of the previous four nodes in its entry-1. Furthermore, as the number of V-state nodes increases, tasks will be transferred to a less preferred node of N_0. The adjusted task transfer–in rates of these nodes are:

$$\begin{aligned}
\tau_6 &= \tau + \gamma_2 \, P(x_6 \leq TH_u) \, P(x_7 \geq TH_f) + \gamma_4 \, P(x_6 \leq TH_u) \, P(x_7 \geq TH_f) \\
&\qquad + \beta_0 \, P(x_6 \leq TH_u) \\
\tau_{10} &= \tau + \gamma_8 \, P(x_{10} \leq TH_u) \, P(x_{11} \geq TH_f) + \beta_0 \, P(x_{10} \leq TH_u) \\
&\quad + \gamma_2 \, P(x_{10} \leq TH_u) \, P(x_{11} \geq TH_f) \, P(x_{14} \geq TH_f) \\
&\quad + \gamma_6 \, P(x_{10} \leq TH_u) \, P(x_{11} \geq TH_f) \, P(x_{12} \geq TH_f) \, P(x_{14} \geq TH_f) \\
\tau_{12} &= \tau + \gamma_8 \, P(x_{12} \leq TH_u) \, P(x_{13} \geq TH_f) \, P(x_{14} \geq TH_f) + \beta_0 \, P(x_{12} \leq TH_u) \\
&\quad + \ (\gamma_4 + \gamma_6 + \gamma_{10}) \ P(x_{12} \leq TH_u) \, P(x_{13} \geq TH_f) \, P(x_{14} \geq TH_f) \\
\tau_3 &= \tau + \gamma_1 \, P(x_3 \leq TH_u) + \gamma_2 \, P(x_3 \leq TH_u) + \beta_0 \, P(x_3 \leq TH_u) \\
&\quad + \ (\gamma_5 + \gamma_9) \ P(x_3 \leq TH_u) \, P(x_7 \geq TH_f) \, P(x_{11} \geq TH_f) \\
&\quad + \ (\gamma_6 + \gamma_{10}) \ P(x_3 \leq TH_u) \, P(x_7 \geq TH_f) \, P(x_{11} \geq TH_f) \, P(x_{15} \geq TH_f) \\
\tau_5 &= \tau + \gamma_4 \, P(x_5 \leq TH_u) + \gamma_1 \, P(x_5 \leq TH_u) \, P(x_7 \geq TH_f) + \beta_0 \, P(x_5 \leq TH_u) \\
&\quad + \gamma_3 \, P(x_5 \leq TH_u) \, P(x_7 \geq TH_f) \, P(x_{13} \geq TH_f) \\
&\quad + \ (\gamma_6 + \gamma_{12}) \ P(x_5 \leq TH_u) \, P(x_7 \geq TH_f) \, P(x_9 \geq TH_f) \times \\
&\qquad P(x_{13} \geq TH_f) \, P(x_{15} \geq TH_f) \\
\tau_9 &= \tau + \gamma_8 \, P(x_9 \leq TH_u) + \gamma_1 \, P(x_9 \leq TH_u) \, P(x_{11} \geq TH_f) \, P(x_{13} \geq TH_f) \\
&\quad + \beta_0 \, P(x_9 \leq TH_u) + (\gamma_3 + \gamma_5 + \gamma_{10} + \gamma_{12}) \, P(x_9 \leq TH_u) \, P(x_{11} \geq TH_f) \times \\
&\qquad P(x_{13} \geq TH_f) \, P(x_{15} \geq TH_f).
\end{aligned}$$

Once the task transfer–in rates of entry-2 nodes are adjusted, the probability of having all entry-1 and entry-2 nodes in no-sharing mode can be calculated as:

$$\begin{aligned}
P(x_1 \geq TH_f, x_2 \geq TH_f, \ldots, x_9 \geq TH_f) &= P(x_1 \geq TH_f) \, P(x_2 \geq TH_f) \\
&\quad \cdots P(x_9 \geq TH_f) \qquad (2.16)
\end{aligned}$$

$$\equiv \varepsilon^{(10)}.$$

Similarly, one can calculate the probability of all other nodes in a 4-cube system being unavailable as $\varepsilon^{(15)} \equiv \varepsilon$.

2.4.2 Approximate Solution

Although q_k's and ε $(= \varepsilon_0)$ can be derived from the upper bound model, it is still very tedious to calculate ε_k, $\forall k > 0$, because there are too many possibilities to consider and each of them is difficult to analyze due to the dependence of LS among the nodes in a buddy set. Moreover, the upper bound model solution fails to include the effects of buddy set size and threshold patterns on the capability of meeting deadlines, while the simulation results in Section 5 did show significant differences when these parameters were changed. So, it is necessary to derive a solution which is simple but closer to the exact solution to Eq. (2.4) than the upper bound solution. Since there are $n!/(n-k)!k!$ possibilities in calculating ε_k in an n-node buddy set, these possibilities can be approximated with only one possibility in which a node is in no-sharing mode with the largest probability. This possibility occurs when all other nodes in the buddy set are in no-sharing mode.

Consider the N_0's preferred list in Fig. 2.2 again. The probabilities of N_2 to N_9 being in no-sharing mode are different from one another due to the adjustment of task transfer–in rates given that more preferred nodes are in no-sharing mode. As the number of no-sharing nodes increases, the adjusted task transfer–in rate of the next preferred node increases. Eventually, N_7, the least preferred node of N_0, will receive the largest number of transfer-in tasks, thus moving it in no-sharing mode with the highest probability within N_0's buddy set. Let P^{sh} and P^{nsh} denote the probabilities of N_7 being in sharing and no-sharing mode, respectively. Then, ε_k's can be approximated by:

$$\varepsilon_k = \frac{n!}{(n-k)!\,k!}\,(P^{nsh})^{n-k}\,(P^{sh})^k. \tag{2.17}$$

Inserting the ε_k's derived from Eq. (2.17) into Eq. (2.5) and applying the iterative method discussed in the previous subsection, we can easily obtain an approximate solution. The calculated results are listed in Tables 2.1 and 2.3 in comparison with the results derived from the upper bound model and simulations (to be discussed in the next section).

Note that the ε_k's derived from Eq. (2.8) are essentially the same as those derived from Eq. (2.7) since both have the same queue state equations for $QL < TH_v$ which are dominant in the probability calculation.

2.5 Performance Analysis

The performance of the LSMSCB is evaluated with the upper bound model, the approximate solution, and simulation. The first two are used to derive the distribution of QL at each node and the probability of meeting deadlines, and analyze the effects of buddy set size, the frequency of state change, and the average system sojourn time of each task. On the other hand, simulation is used to verify the analytic results.

2.5.1 Analytic Results

The proposed model can be applied to any arrival process, but the transition probability α_{k_t} must be given prior to the calculation of q_k's with Eqs. (2.4) and (2.8). To demonstrate the main idea of the LSMSCB, we present some numerical results for the case when both arrivals of external and transferred–in tasks follow exponential distributions. (Note, however, that the LSMSCB and models are not restricted to exponential distributions.)

Distribution of Queue Length

The distributions of QL for two different external task arrival rates in a 16–node system are calculated with the upper bound model and the approximate solution, and compared with simulation results as well as with the case of no LS (Table 2.1). The q_k's calculated with the upper bound solution and the approximate solution are very close to each other when $k \leq TH_v$. This was expected because the two differ only when $k > TH_v$. This fact also ensures the accuracy in calculating ε, since it was computed with the q_k's derived from the upper bound model and then used to derive approximate q_k's from Eq. (2.4). Moreover, the distribution of QL obtained via simulation is shown to be very close to the approximate solution for all k and is bounded by the upper bound solution when $k > TH_v$. Since the approximate solution is always very close to the exact solution, we will use it in the following discussions unless stated otherwise.

Probability of Meeting Deadlines

A task is said to be *missed* if its system sojourn time[4] exceeds a given deadline. According to the queueing model, the completion time of a newly arriving task is equal to the current queue length plus one unit of time. Since the probability of $QL > TH_v$ is quite small, one can choose TH_v to be one less than the given deadline such that the probability of missing deadlines, or simply called the *missing probability*, becomes the probability of encountering $QL > TH_v$ at the time of a task arrival. Clearly, the missing probability depends on the given deadline and system load. However, by selecting a proper threshold pattern and a buddy set size, it is possible to minimize the missing probability. Figs. 2.4 and 2.5 are the plots of missing probabilities vs. task deadlines for different threshold patterns.

Generally, the missing probability increases as the system load gets heavier (Fig. 2.6) and/or the deadline gets shorter. By choosing an appropriate threshold pattern, e.g., "1 2 3" in Figs. 2.4–2.6, the missing probability can be reduced to a small value even when system load fluctuates (except when the system is overloaded, e.g., $\lambda \geq 0.9$). The analytic results also show that the choice of a threshold pattern is sensitive to system load. For ex-

[4]The system sojourn time of a task is composed of its execution time, queueing time, and task transfer time.

ample, threshold pattern "0 1 1" results in a small missing probability when the system is underloaded, while resulting in a much higher missing probability as system load increases. Threshold pattern "1 2 3" is found to yield a reasonably small missing probability for a wide range of load density ($0 < \lambda < 0.8$). Fig. 2.4 shows an interesting result of the upper bound model: missing probabilities for different threshold patterns are quite close to each other, and thus, difficult to tell which pattern is better over the others. This is opposite to what has been shown by the approximate solution in Figs. 2.5 and 2.6. That is, the upper bound model exaggerates the probability of switching to a queue length greater than TH_v, and thus, the effect of threshold pattern becomes insignificant. Since threshold pattern "1 2 3" exhibits the best performance among the three patterns considered, the performance with this pattern is further compared with simulation results. As shown in Figs. 2.7 and 2.8, the missing probability obtained from the simulation is always upper bounded by those obtained from the upper bound model and is very close to the approximate solution.

Average System Sojourn Time vs. Missing Probability

The average system sojourn time can be obtained by dividing the sum of all tasks' system sojourn times by the total number of tasks processed. Mathematically, the average system sojourn time is equal to the expected task execution time, $\sum_{k=0}^{\infty}(k+1)q_k$. The average system sojourn time is calculated for several different threshold patterns and buddy set sizes as presented in Table 2.2. One interesting result is that the lower TH_u and TH_v, the smaller the average system sojourn time results, and that buddy set size has only minor effects on the average system sojourn time. This is in sharp contrast with the results reported in [40], where the average system sojourn time under the shortest queue policy was shown to be only slightly smaller than that under the threshold policy. In the LSMSCB, the shortest queue (threshold) policy is equivalent to selecting $TH_u = 0$ and $TH_f = 1$ ($TH_u > 0$). As shown in Table 2.2, the threshold pattern with $TH_u = 0$ and $TH_f = 1$

always results in a substantially smaller average system sojourn time than the pattern with $TH_u > 0$. This is the advantage resulting from the state-change broadcasts since the traffic overhead for collecting state information in the case of $TH_u = 0$ is essentially the same as the case of $TH_u > 0$. However, the traffic overhead associated with the shortest queue policy is higher than that of the threshold policy due to its required probing of other nodes [40], offsetting the potential gain to be made by transferring tasks to a node with the shortest queue. Consequently, the LSMSCB outperforms other sender–initiated LS algorithms even when the average system time is used to measure their performance.

Another important result is that a threshold pattern that results in a lower average system sojourn time does not always yield a lower missing probability. For example, consider the buddy sets of size 10 in Tables 2.2 and 2.3. Pattern "0 1 2" results in a smaller average task system sojourn time than "1 2 2", but a larger missing probability than "1 2 2" when the deadline is greater than 2. Moreover, some thresholds may result in almost the same average task system sojourn time but yield quite different missing probabilities, e.g., "0 2 3" and "1 2 3" when the deadline is greater than 3 in Table 2.3. Hence, those approaches based on minimizing the average task system sojourn time alone may not be applicable to the analysis of real-time systems.

System Utilization

The system utilization is defined as the ratio of external task arrival rate (λ) to the system service rate, which is unity in the LSMSCB model. (Thus, the system utilization is simply λ.) Since the missing probability depends on system workload (Fig. 2.8), we can solve Eq. (2.8) to derive λ as a function of q_k's and then the maximum system utilization can be obtained by equating q_{TH_v+1} to the specified missing probability. Some of calculated results are plotted in Fig. 2.9. This is in sharp contrast to the common notion that the real-time systems have to be designed to sacrifice utilization for a lower missing probability.

Buddy Set Size and Preferred List

The effect of changing buddy set size on the missing probability can best be explained by the approximate solution as shown in Fig. 2.10. Buddy set size affects the missing probability significantly when it grows from 4 to 10 and $\lambda > 0.7$, but its incremental effect becomes insignificant when buddy set size is greater than 10. Actually, there is little notable decrease in the missing probability when buddy set size grows beyond 15. Surprisingly, the missing probability for a 4-node buddy set is about three orders of magnitude less than that without LS when the system is underloaded ($\lambda \leq 0.5$), and is about the same as those for buddy sets of size larger than 10 when the system is overloaded ($\lambda > 0.8$). So, buddy set size can be chosen to range from 10 to 15, regardless of the system size. The most interesting result is found to be that the missing probability in a large system (of 64 nodes in Table 2.4) is much smaller than that of a small system (of 16 nodes in Table 2.3). For example, consider threshold "1 2 3" with a 10-node buddy set at $\lambda = 0.8$. The missing probability of a 64-node system is about 3, 4, and 20 times smaller than that of the 16-node system when the deadline is 4, 5, and 6, respectively. This significant improvement was found for all other threshold patterns, thus indicating that the larger the system size, the better the performance of the proposed LS method will result. (See the next paragraph for a reasoning about this.) Note that the traffic overhead for broadcasting state changes remains unchanged and independent of system size because the buddy set size is fixed (to 10–15). Furthermore, the incremental decrease in missing probability becomes insignificant when the buddy set size is over 15 (Table 2.4).

Use of buddy sets and preferred lists in the LSMSCB plays a major role in lowering the missing probability for a large system. As discussed in Section 2.3, the buddy set of a node consists of those nodes in its physical proximity, and each node in the buddy set is selected according to the order of its preference. Moreover, preferred lists are constructed in such a way that each node is the i^{th} ($i = 1, \ldots, n$) preferred node of only one other node and the preferred lists of the nodes in the same buddy set are completely different from each other. As a result, the overflow tasks within each buddy set will be evenly shared by all underloaded nodes in the entire system, rather than overloading a few underloaded nodes within the same buddy set. As the system size grows, the percentage of common nodes in the preferred lists of a buddy set gets smaller, and thus, the overflow tasks are more evenly distributed in the system, resulting in a better performance.

Frequency of State Change

In the LSMSCB, each node needs to broadcast change of state to all the other nodes in its buddy set. Since a state change occurs when a node switches from U–state to F–state and vice versa, the probability of a state change becomes: $P(x_{k_t+1} \leq TH_u | x_{k_t} \geq TH_f)$ $+P(x_{k_t+1} \geq TH_f | x_{k_t} \leq TH_u)$. The computation results of Table 2.5 showed that the frequency of state change can be reduced to 10 – 15% of the total number of arrived tasks by setting $TH_f - TH_u = 2$. Note that this frequency becomes about 100% of the number of external task arrivals when $TH_u = 0$ and $TH_f = 1$. The resulting high frequency of state change should rule out this type of threshold patterns.

The traffic overhead for collecting state information in the LSMSCB is determined by the frequency of state change and buddy set size, while it was determined by the number of task transfers and probing in [40]. Since the frequency of state change can be controlled by adjusting the difference between TH_u and TH_f, this frequency with threshold "1 3 3" and $\lambda > 0.7$ is found to be about the same as the percentage of external arrivals that are transferred out. Moreover, transferring one task may require the probes of 5 to 6 other nodes [40] and each probe generates two communication messages (one for request and the other for response) in sender–initiated methods, whereas each state–change broadcast in the LSMSCB generates n messages, where n is the buddy set size. The traffic overhead for broadcasting state changes for threshold "1 3 3" and a 10-node buddy set is about the same as that in a sender–initiated approach. However, the time for selecting a destination node in this method is much smaller than that in any sender–initiated approach, because in this approach, a task can be transferred upon its arrival without probing any other nodes. Besides, each task transfer in the LSMSCB will take less time than other LS methods, because use of a preferred list will usually locate a receiver in the sender's physical proximity.

Delays in Task Transfer and State–Change Broadcasts

When a node selects and transfers a task to a U–state node, the U–state node may receive an external task and switch to V–state before the transferred task arrives. In this case, the transferred task will actually arrive at a V–state node. Thus, the probability of transition to V–state with non-zero task arrivals (i.e., α_k for $k > 0$) is larger than that used in Eq. (2.5), where transferred tasks are assumed to be accepted only when a receiving node is in U–state. One can estimate this probability and adjust the corresponding α_k's. The broadcasting delay has the same effect as the task transfer delay.

Although these delays may affect the distribution of QL, the missing probability can be made insensitive to them by properly choosing a threshold pattern. For example, a task which arrives when $QL = TH_f$ will not be transferred again if $TH_v > TH_f$, e.g., threshold "1 2 3", but it will be retransferred if $TH_v = TH_f$, e.g., threshold "1 2 2". Since task retransfers induce traffic overheads without improving the capability of meeting deadlines, the threshold patterns that are sensitive to these delays may result in a higher missing probability than those that are not. The effect of these delays on the missing probability is investigated further in the simulation.

2.5.2 Simulation Results

For the simulation the average load density is varied from 0.5 to 0.9, and buddy sets of sizes 4, 10, and 15 are considered. Ten threshold patterns are chosen out of all possible combinations for the simulation of a 4-cube system and the results are given in all tables except for Table 2.2. A few selected thresholds for a 6-cube system are also simulated and given in Table 2.4. The time for transferring a task between two nodes within a buddy set is assumed to be 10% of the task execution time and the time for informing a state change to one of the nodes in a buddy set is assumed to be 1% of the task execution time.

In most cases, simulation results are consistent with, and close to, the approximate solution. However, the analytically derived q_k's for $k > TH_v$ are always less than those obtained from the simulation when the system is underloaded ($\lambda \leq 0.5$) or overloaded ($\lambda \geq 0.9$), especially in the threshold patterns with $TH_u > 0$. This discrepancy may have been caused by the delays in transferring tasks and broadcasting state changes. The effect of setting TH_v to be larger than TH_f is also observed in the simulation. The percentage of task retransfers for the case of $TH_f = TH_v$ is higher than that for the case of $TH_f < TH_v$. Hence, the threshold pattern with $TH_f < TH_v$ is a better choice than the pattern with $TH_f = TH_v$. This observation also explains why the missing probability associated with threshold "1 2 3" is smaller than that of "1 2 2" when $\lambda > 0.7$, deadline > 3, and buddy set size > 10.

To study the effect of changing task transfer costs, we ran simulations with task transfer costs 5, 10, 20, and 30% of the task execution time. As shown in Table 2.6, the missing probability of threshold "1 2 3" remains almost unchanged. Based on all the above results, it is concluded that threshold "1 2 3" is good for a wide range of system load. Note that, although the missing probability of threshold "1 2 2" is usually close to that of threshold "1 2 3", the task transfer rate associated with "1 2 2" is much higher than that with "1 2 3". Thus, considering cost-performance effectiveness, threshold "1 2 3" is a better choice than "1 2 2".

2.5.3 Advantages of Using Analytic Approaches

There are several advantages of using the upper bound model and the approximate solution, as compared to simulations. First, the results derived from the upper bound model can be used to guarantee the specified system reliability, because the actual missing probability is always less than that derived from the upper bound model. Second, system utilization can be analyzed by using the analytic models. Third, the analytic models provide, at almost no cost, many pieces of useful information with accuracy. For example, any meaningful simulation of the LSMSCB requires hundreds of CPU hours (in a computer as powerful as VAX-11/780) to get an accuracy of 10^{-6} in the calculation of q_k's for a system of moderate size. Moreover, simulation may be able to provide information only for a particular system workload; it is too costly to generate q_k's with simulation as a function of system workload.

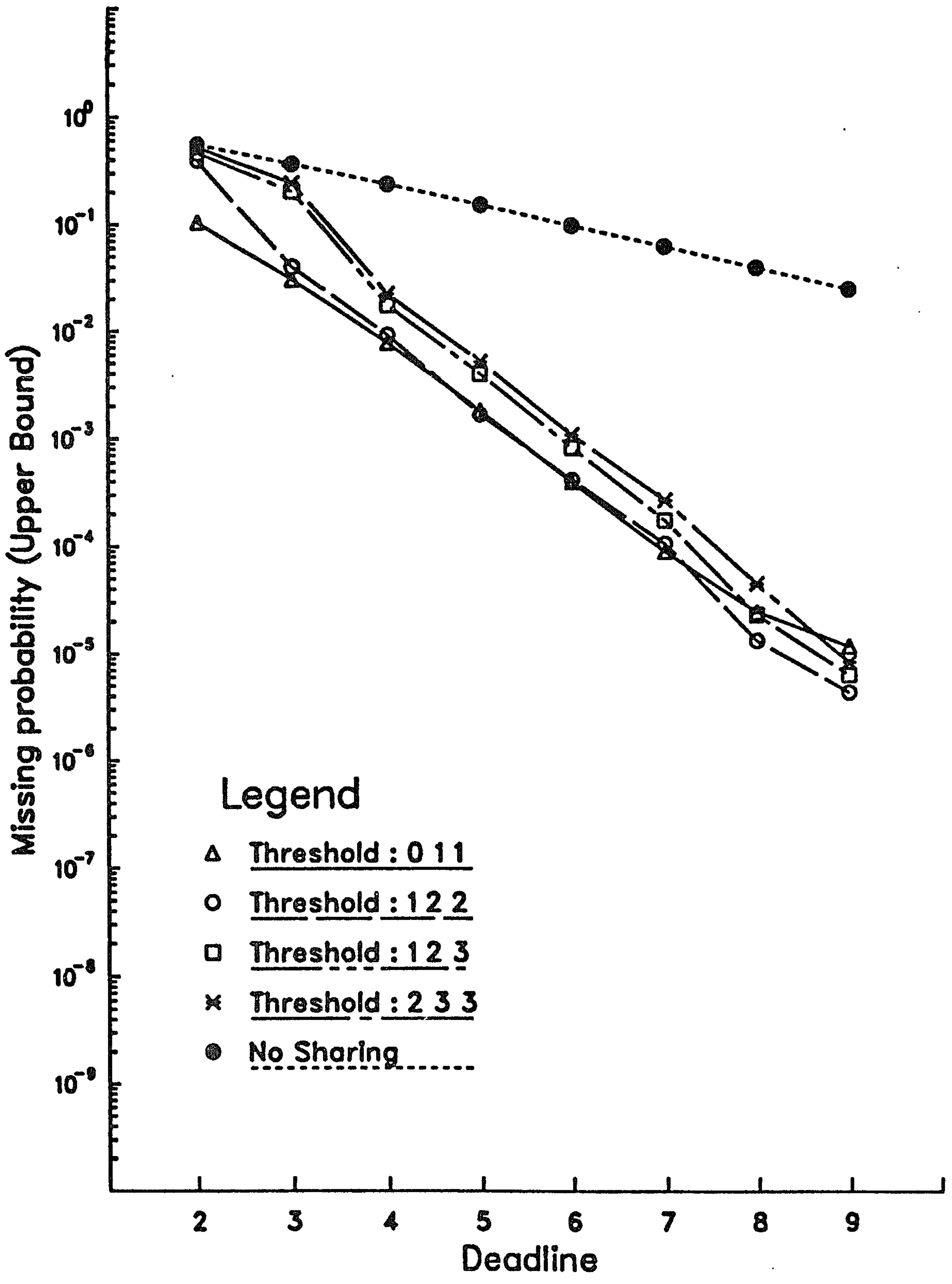


Figure 2.4: Upper bound missing probabilities vs. deadlines for different thresholds when $\lambda = 0.8$.

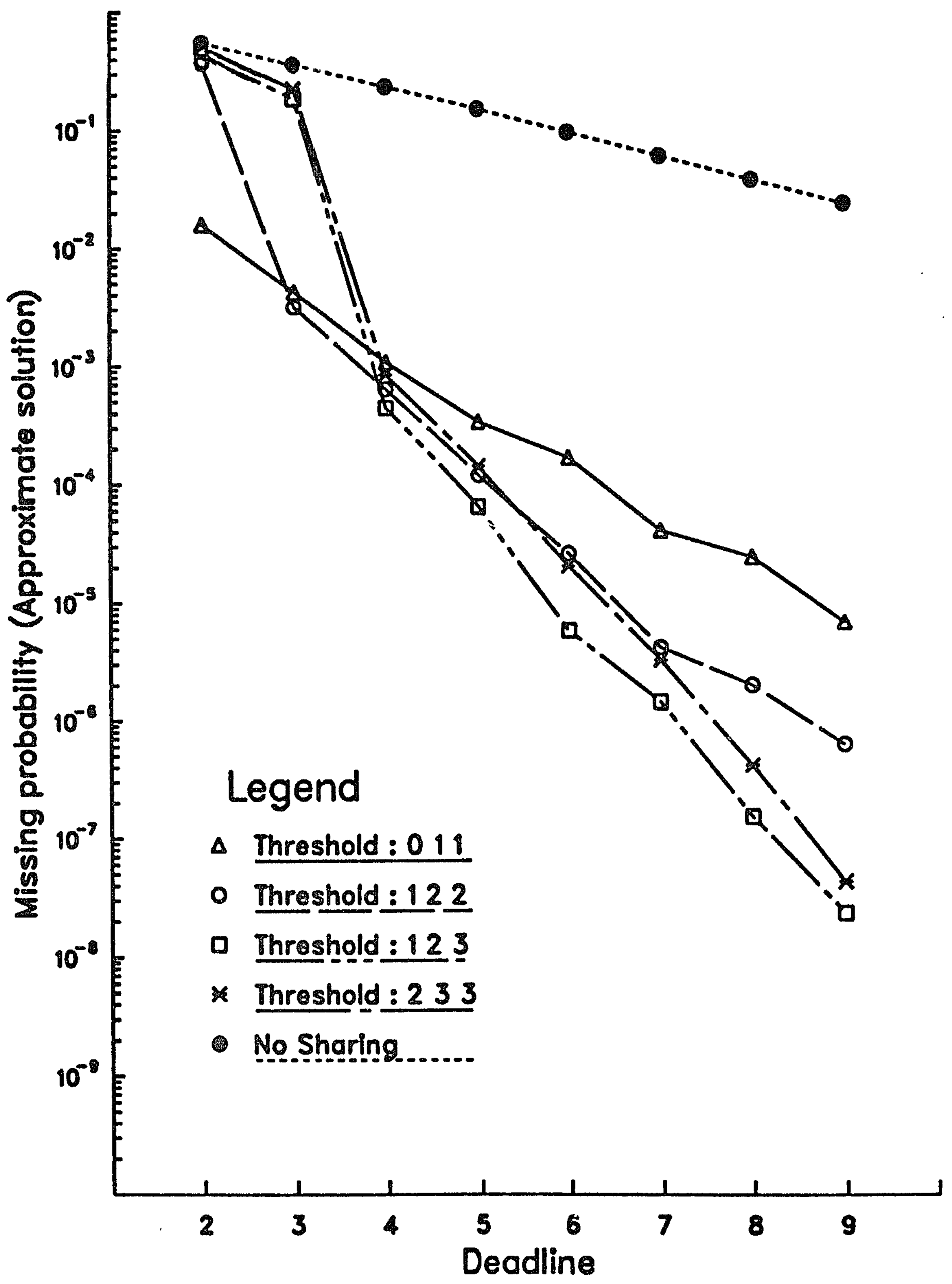


Figure 2.5: Approximate missing probabilities vs. deadlines for different thresholds when $\lambda = 0.8$.

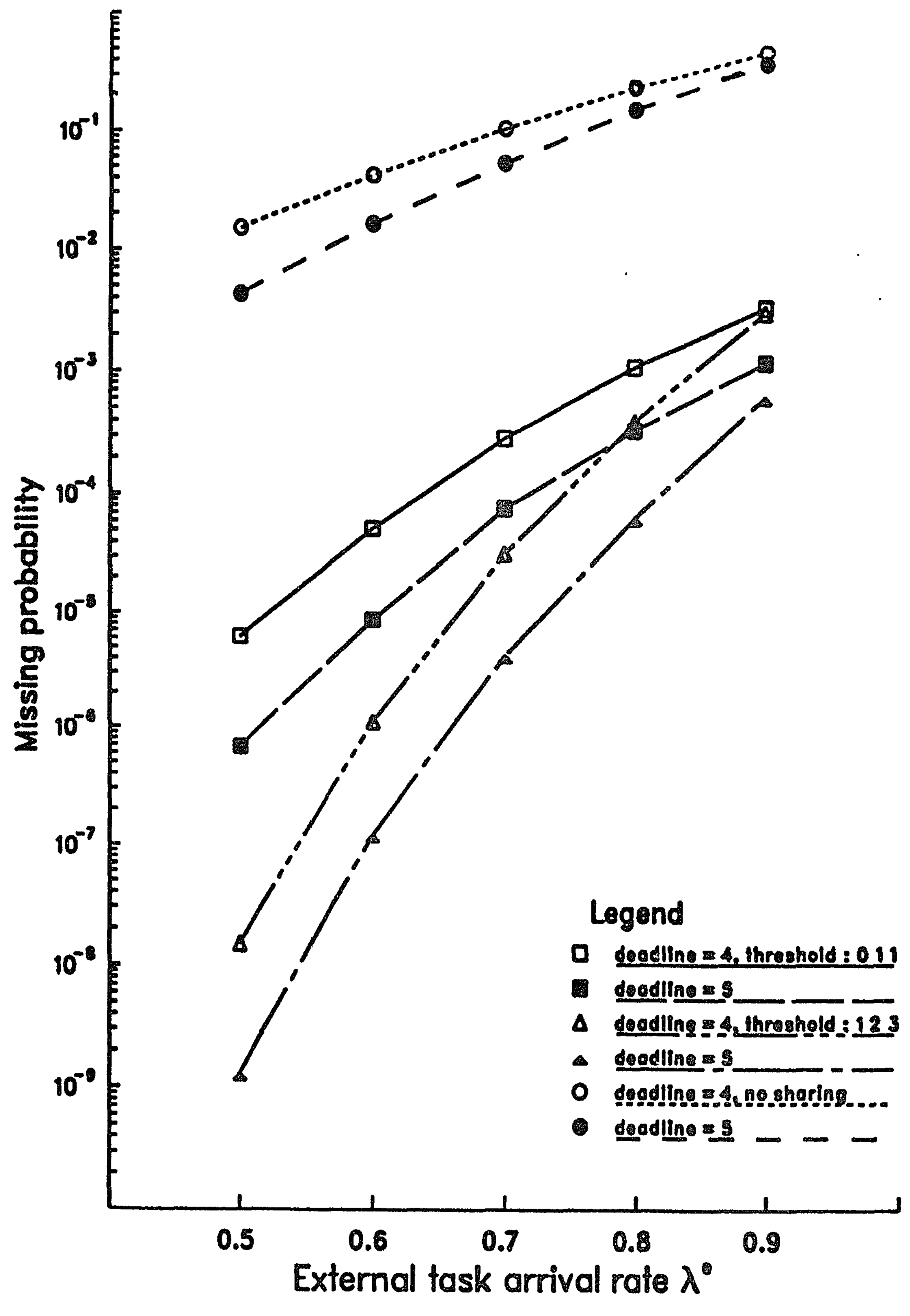


Figure 2.6: Approximate missing probabilities vs. load density for different thresholds and deadlines.

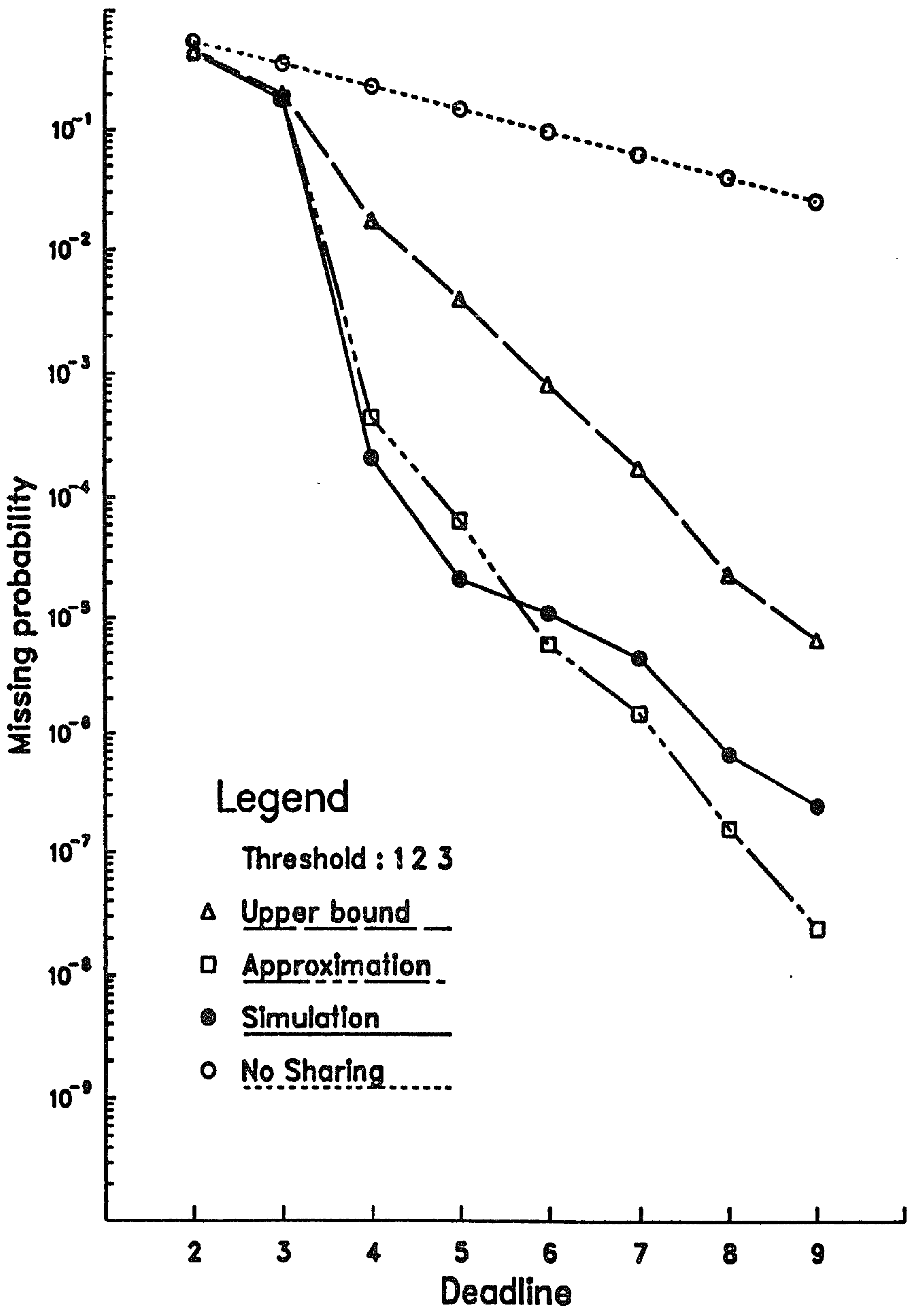


Figure 2.7: Simulated, approximate and upper bound missing probabilities vs. deadlines in when $\lambda = 0.8$.

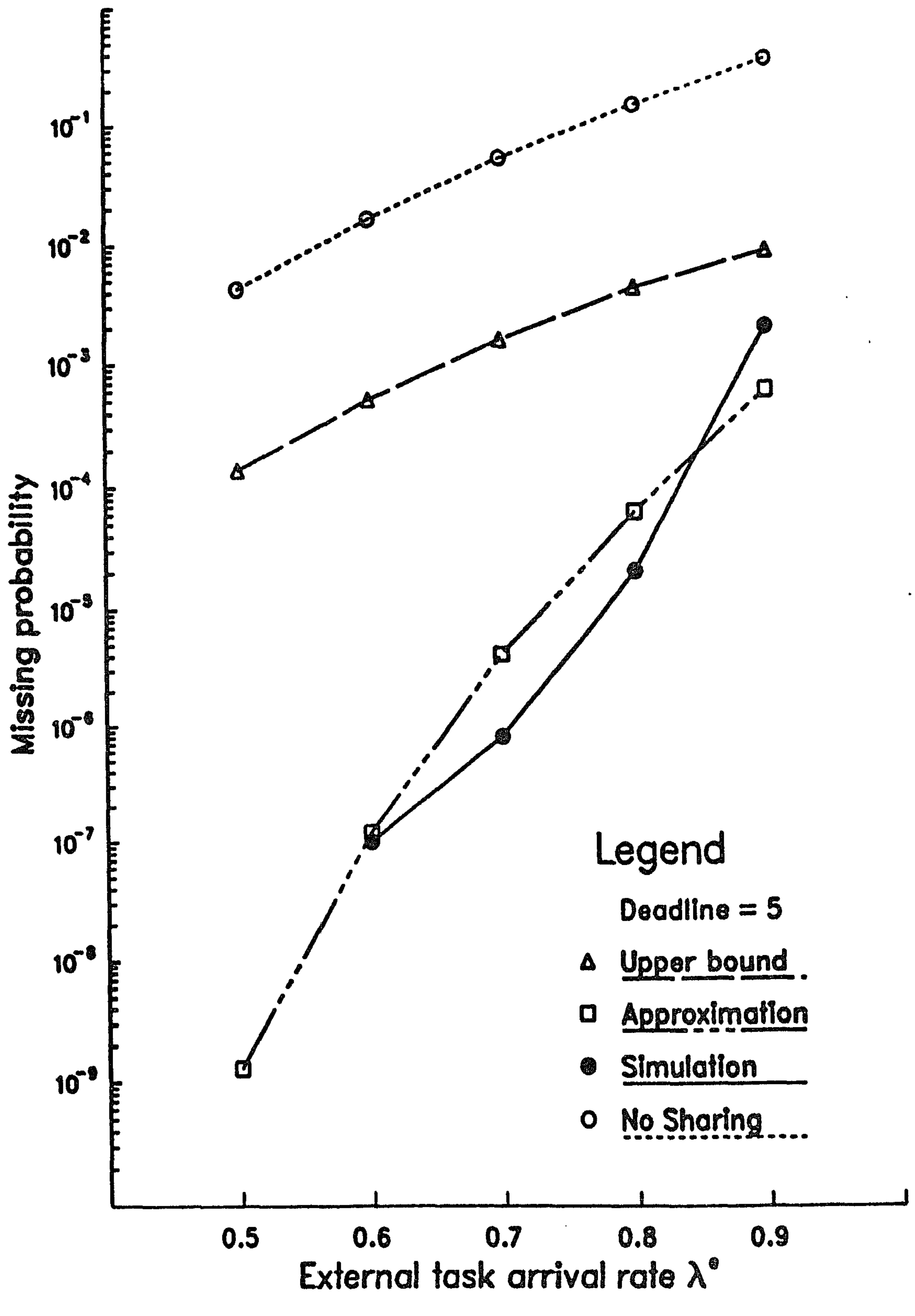


Figure 2.8: Approximate, upper bound, and simulated missing probabilities vs. load density with threshold pattern "1 2 3".

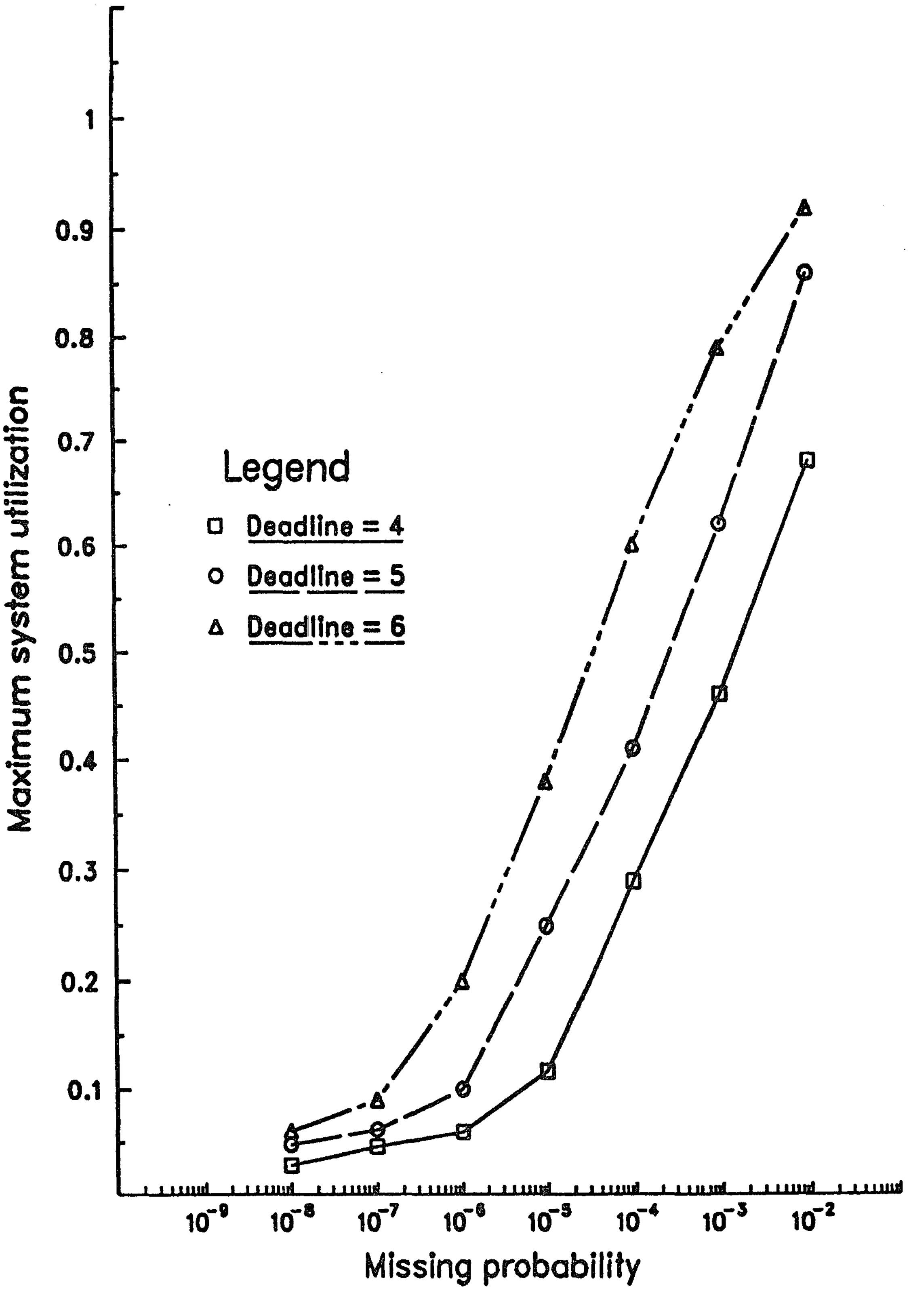


Figure 2.9: Maximum system utilization vs. missing probability for different deadlines.

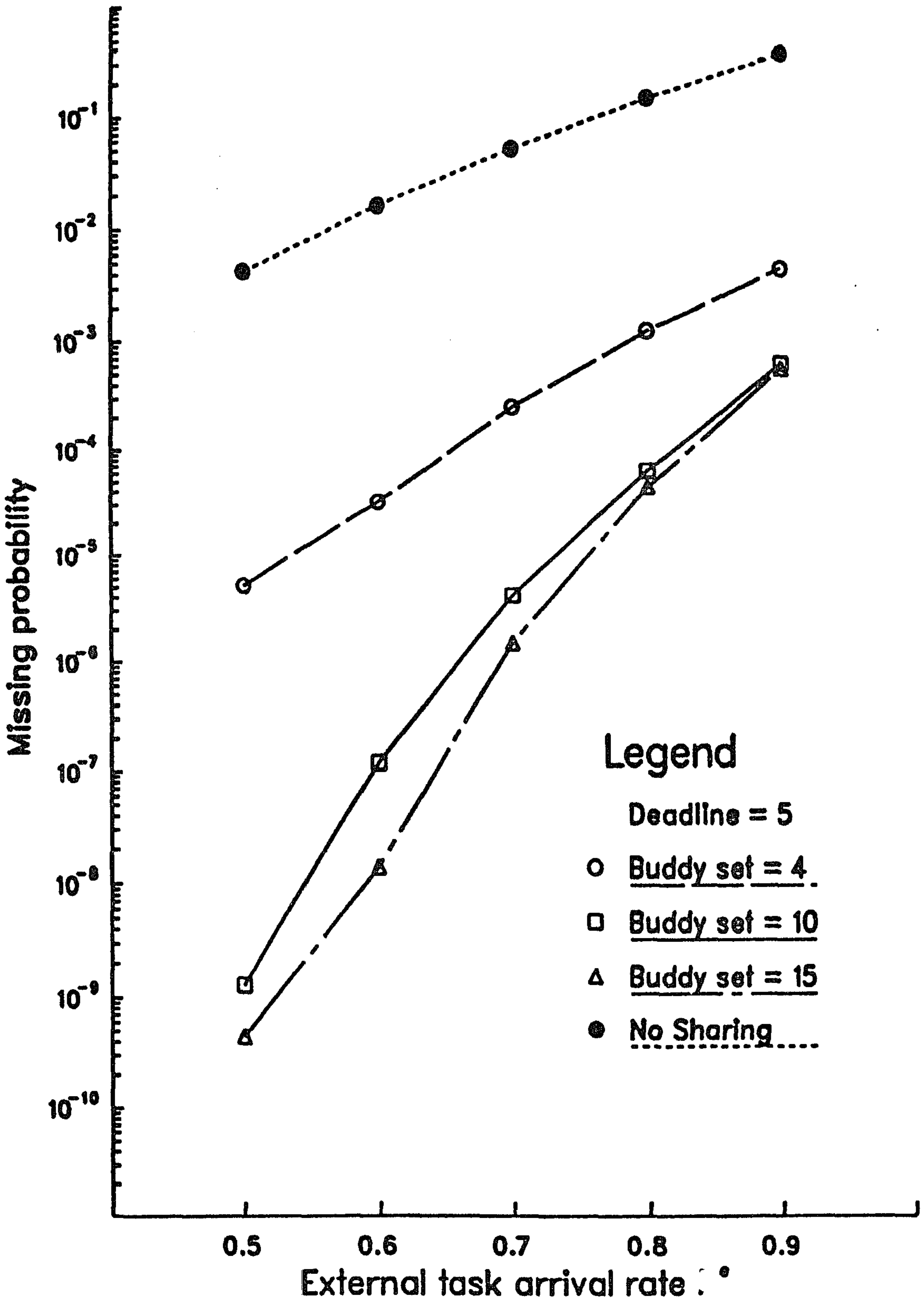


Figure 2.10: Approximate missing probabilities vs. load density with threshold "1 2 3" and different buddy set sizes.

($\lambda^e = 0.5$) Model Queue Length	Simulation	Approximation	Upper bound	no load sharing
0	0.5037	0.4987	0.4992	0.5000
1	0.3316	0.3317	0.3321	0.3244
2	0.1254	0.1289	0.1278	0.1226
3	0.0387	0.0406	0.0398	0.0377
4	3.13×10^{-7}	1.43×10^{-8}	0.0010	0.0109
5	$< 10^{-7}$	1.21×10^{-9}	0.0001	0.0031
6	$< 10^{-7}$	9.09×10^{-11}	1.3×10^{-5}	0.0009
7	$< 10^{-7}$	5.96×10^{-12}	1.4×10^{-6}	0.0003
8	$< 10^{-7}$	3.35×10^{-13}	1.33×10^{-7}	7.16×10^{-5}
9	$< 10^{-7}$	1.46×10^{-14}	1.28×10^{-8}	2.04×10^{-5}

($\lambda^e = 0.8$) Model Queue Length	Simulation	Approximation	Upper bound	no load sharing
0	0.2213	0.2264	0.2185	0.2004
1	0.3317	0.3194	0.3136	0.2456
2	0.2656	0.2675	0.2651	0.1898
3	0.1810	0.1862	0.1853	0.1278
4	0.0002	0.0003	0.0135	0.0834
5	2.01×10^{-5}	5.94×10^{-5}	0.0032	0.0542
6	8.98×10^{-6}	8.99×10^{-6}	0.0007	0.0353
7	3.21×10^{-6}	1.18×10^{-6}	0.0001	0.0229
8	7.81×10^{-7}	1.36×10^{-7}	1.86×10^{-5}	0.0149
9	3.90×10^{-7}	1.41×10^{-8}	2.38×10^{-6}	0.0097

Table 2.1: Comparison of distributions of queue length from analytical and simulation results with threshold pattern = "1 2 3", the number of nodes =16 and buddy set size =10.

($\lambda^e = 0.5$) Buddy set Threshold	4	10	15
0 1 1	1.414	1.394	1.401
0 1 2	1.609	1.602	1.605
0 2 2	1.608	1.608	1.622
0 2 3	1.696	1.695	1.700
1 2 2	1.618	1.618	1.625
1 2 3	1.698	1.698	1.701
1 3 3	1.700	1.700	1.703
2 3 3	1.701	1.701	1.703
No sharing	1.750	1.750	1.750

($\lambda^e = 0.8$) Buddy set Threshold	4	10	15
0 1 1	1.918	1.692	1.651
0 1 2	2.236	2.065	1.033
0 2 2	2.120	2.075	2.071
0 2 3	2.426	2.382	2.380
1 2 2	2.123	2.109	2.110
1 2 3	2.423	2.407	2.405
1 3 3	2.427	2.422	2.421
2 3 3	2.475	2.472	2.473
No sharing	3.370	3.370	3.370

Table 2.2: Average task system sojourn time.

$(\lambda^e = 0.5)$	Threshold / Deadline	0 1 1	0 1 2	1 2 2	1 2 3	2 3 3	No Sharing
simulation	2	0.0007	0.1296	0.1339	0.1642	0.1661	—
	3	4.81×10^{-5}	8.82×10^{-5}	1.73×10^{-6}	0.0388	0.0392	—
	4	3.10×10^{-6}	5.32×10^{-6}	1.46×10^{-7}	3.13×10^{-7}	6.59×10^{-7}	—
	5	2.85×10^{-7}	$< 10^{-7}$	$< 10^{-7}$	$< 10^{-7}$	$< 10^{-7}$	—
	6	$< 10^{-7}$	$< 10^{-7}$	$< 10^{-7}$	$< 10^{-7}$	$< 10^{-7}$	—
analytic (approximation)	2	0.0004	0.1370	0.1446	0.1681	0.1691	0.1756
	3	5.10×10^{-5}	8.41×10^{-5}	7.46×10^{-7}	0.0398	0.0406	0.0530
	4	6.25×10^{-6}	9.73×10^{-6}	7.11×10^{-8}	1.46×10^{-8}	1.87×10^{-9}	0.0152
	5	6.81×10^{-7}	9.51×10^{-7}	6.01×10^{-9}	1.31×10^{-9}	1.44×10^{-10}	0.0043
	6	7.61×10^{-8}	8.8×10^{-8}	9.11×10^{-11}	1.31×10^{-11}	1.01×10^{-11}	0.0012

$(\lambda^e = 0.8)$	Threshold / Deadline	0 1 1	0 1 2	0 2 3	1 2 2	1 2 3	2 3 3	No Sharing
simulation	2	0.0745	0.3258	0.4414	0.3461	0.4468	0.4858	—
	3	0.0176	0.0305	0.1805	0.0019	0.1812	0.2007	—
	4	0.0045	0.0067	0.012	0.0003	0.0002	0.0002	—
	5	0.0014	0.0015	0.0003	8.37×10^{-5}	2.11×10^{-5}	2.28×10^{-5}	—
	6	0.0006	0.0004	0.0001	3.61×10^{-5}	1.11×10^{-5}	6.18×10^{-6}	—
analytic (approximation)	2	0.0161	0.3315	0.4342	0.3723	0.4541	0.5151	0.5539
	3	0.0043	0.0105	0.1772	0.0032	0.1866	0.2256	0.3641
	4	0.0011	0.0024	0.0013	0.0007	0.0004	0.0009	0.2363
	5	0.0004	0.0005	0.0003	0.0001	6.54×10^{-5}	0.0002	0.1528
	6	0.0002	0.0002	6.95×10^{-5}	2.64×10^{-5}	1.2×10^{-5}	2.58×10^{-5}	0.0986

Table 2.3: Missing Probabilities for various threshold patterns and task deadlines when $\lambda^e = 0.5$ and $\lambda^e = 0.8$, and buddy set size = 10.

$(\lambda^e = 0.8)$ Buddy set	Threshold Deadline	0 1 2	1 2 2	1 2 3	1 3 3	2 3 3
4	2	0.3702	0.3491	0.4506	0.4555	0.4873
	3	0.0950	0.0071	0.1856	0.1885	0.2018
	4	0.0320	0.0018	0.0047	0.0041	0.0011
	5	0.0107	0.0004	0.0008	0.0006	0.0002
	6	0.0034	9.1×10^{-5}	0.0002	0.0002	5.8×10^{-5}
10	2	0.3181	0.3448	0.4463	0.4526	0.4856
	3	0.0229	0.0012	0.1806	0.1860	0.2008
	4	0.0043	8.9×10^{-5}	6.4×10^{-5}	0.0004	0.0002
	5	0.0008	8.1×10^{-6}	5.2×10^{-6}	1.6×10^{-5}	6.6×10^{-6}
	6	0.0001	6.4×10^{-7}	4.7×10^{-7}	2.9×10^{-6}	5.8×10^{-7}
15	2	0.3073	0.3448	0.4463	0.4532	0.4856
	3	0.0089	0.0012	0.1806	0.1863	0.2006
	4	0.0013	5.7×10^{-5}	3.9×10^{-5}	0.0004	0.0002
	5	0.0002	5.1×10^{-6}	3.2×10^{-6}	1.1×10^{-5}	7.6×10^{-6}
	6	2.0×10^{-5}	2.4×10^{-7}	1.6×10^{-7}	8.4×10^{-7}	2.6×10^{-7}
21	2	0.3026	0.3445	0.4463	0.4526	0.4856
	3	0.0033	0.0012	0.1806	0.1860	0.2008
	4	0.0004	6.0×10^{-5}	3.7×10^{-5}	0.0004	0.0002
	5	9.2×10^{-5}	4.1×10^{-6}	2.2×10^{-6}	1.1×10^{-5}	4.6×10^{-6}
	6	1.1×10^{-5}	1.7×10^{-7}	$< 10^{-7}$	7.4×10^{-7}	2.9×10^{-7}

Table 2.4: Missing Probabilities in a 6-cube for different thresholds and buddy set sizes.

($\lambda^e = 0.5$)	Transferred Tasks		Frequency of State Change	
Threshold	Simulation	Analytic	Simulation	Analytic
0 1 1	0.1071	0.1300	1.0792	1.1128
0 1 2	0.0304	0.0330	1.0203	1.0266
0 2 2	0.0484	0.0501	0.1327	0.1471
1 2 2	0.0332	0.0392	0.1574	0.1754
1 2 3	0.0092	0.0098	0.1511	0.1552
1 3 3	0.0097	0.0101	0.0407	0.0370
2 3 3	0.0089	0.0100	0.0472	0.0487

($\lambda^e = 0.8$)	Transferred Tasks		Frequency of State Change	
Threshold	Simulation	Analytic	Simulation	Analytic
0 1 1	0.2241	0.2230	0.6277	0.8118
0 1 2	0.1145	0.1600	0.5283	0.5984
0 2 2	0.2274	0.2015	0.1613	0.2531
1 2 2	0.1677	0.1891	0.2576	0.3316
1 2 3	0.0877	0.0811	0.2182	0.2404
1 3 3	0.1064	0.0934	0.1050	0.1204
2 3 3	0.0875	0.1050	0.1646	0.1782

Table 2.5: Number of task transfers vs. frequency of state change for different thresholds.

$(\lambda^e = 0.8)$ Transfer Cost	Threshold Deadline	0 1 2	1 2 2	1 2 3	2 3 3
5%	2	0.3227	0.3460	0.4473	0.4860
	3	0.0307	0.0013	0.1812	0.2007
	4	0.0067	0.0002	0.0003	0.0001
	5	0.0015	6.51×10^{-5}	4.26×10^{-5}	1.65×10^{-5}
	6	0.0004	2.83×10^{-5}	9.11×10^{-6}	5.98×10^{-6}
10%	2	0.3258	0.3461	0.4468	0.4858
	3	0.0305	0.0019	0.1812	0.2007
	4	0.0067	0.0003	0.0002	0.0002
	5	0.0015	8.37×10^{-5}	2.11×10^{-5}	2.28×10^{-5}
	6	0.0004	3.61×10^{-5}	1.11×10^{-5}	6.18×10^{-6}
20%	2	0.3300	0.3530	0.4504	0.4847
	3	0.0295	0.0046	0.1835	0.2017
	4	0.0065	0.0005	0.0004	0.0007
	5	0.0015	8.69×10^{-5}	5.58×10^{-5}	5.31×10^{-5}
	6	0.0004	3.8×10^{-5}	1.29×10^{-5}	6.33×10^{-6}
30%	2	0.3488	0.3817	0.4681	0.4914
	3	0.0320	0.0168	0.1956	0.2177
	4	0.0072	0.0023	0.0016	0.0050
	5	0.0016	0.0003	0.0002	0.0005
	6	0.0004	4.34×10^{-5}	2.93×10^{-5}	4.95×10^{-5}

Table 2.6: Missing Probabilities vs. task transfer costs for different deadlines and thresholds.

CHAPTER 3

OPTIMIZATION OF THE LSMSCB

3.1 Introduction

The numerical solutions for QL derived in Section 2.5 showed only a few examples and it is practically too tedious to derive the numerical results for all threshold patterns and buddy set sizes. It is therefore important to derive a closed-form expression for the distribution of QL to characterize the behavior of a real–time system. Based on this closed-form distribution, 'optimal' threshold patterns and optimal buddy set sizes can be determined either by minimizing the communication overhead — such as the frequency of collecting state information and the number of task transfers — incurred by the LSMSCB while keeping P_{dyn} below any required level, or by minimizing P_{dyn} while keeping communication cost below any given level [59]. An upper bound of processor utilization can also be derived while reducing P_{dyn} to any given level.

The derivation of the approximate closed-form solution is discussed in Section 3.2, and the optimization of LSMSCB is treated in Section 3.3.

3.2 Approximate Closed–Form Solution

Eq. (2.5) cannot be solved in one step for the following two reasons. First, α^*'s in these equations depend on the task transfer-in rate, τ, which in turn depends on q_k's. Second, since q_k's for $k > v$ depend on those q_k's for $k \leq v$, it is impossible to obtain the distribution of queue lengths in one step. Since the probability of $QL > v$ is very small, Eq. (2.5) can be divided into two parts which are then solved separately: q_k's for $0 \leq k \leq v$ and q_k's for $k > v$.

3.2.1 Solving q_k for $0 \leq k \leq v$

Eq. (2.5) can be rewritten as:

$$\begin{aligned}
q_1 &= \frac{1-\alpha_0^*}{\alpha_0^*} q_0 \\
\frac{\alpha_1^*-1}{\alpha_0^*} q_1 + q_2 &= -\frac{\alpha_1^*}{\alpha_0^*} q_0 \\
\frac{\alpha_2^*}{\alpha_0^*} q_1 + \frac{\alpha_1^*-1}{\alpha_0^*} q_2 + q_3 &= -\frac{\alpha_2^*}{\alpha_0^*} q_0 \\
&\vdots \qquad\qquad (3.1)\\
\frac{\alpha_{u-1}^*}{\alpha_0^*} q_1 + \frac{\alpha_{u-2}^*}{\alpha_0^*} q_2 + \cdots + \frac{\alpha_2^*}{\alpha_0^*} q_{u-2} + \frac{\alpha_1^*-1}{\alpha_0^*} q_{u-1} + q_u &= -\frac{\alpha_{u-1}^*}{\alpha_0^*} q_0 \\
&\vdots \\
\frac{\alpha_{v-1}^*}{\alpha_0} q_1 + \frac{\alpha_{v-2}^*}{\alpha_0^*} q_2 + \cdots + \frac{\alpha_{v-u}^*}{\alpha_0} q_u + \cdots + \frac{\alpha_1-1}{\alpha_0} q_{v-1} + q_v &= -\frac{\alpha_{v-1}^*}{\alpha_0} q_0 .
\end{aligned}$$

Eq. (3.1) can also be expressed in vector-form as $A_1 Q_1 = C q_0$, where A_1 is a $v \times v$ lower triangular matrix, $Q_1 = [q_1 \; q_2 \; \cdots q_u \cdots q_v]^T$, and

$$C = \left[\frac{1-\alpha_0^*}{\alpha_0^*} \;\; -\frac{\alpha_1^*}{\alpha_0^*} \;\; -\frac{\alpha_2^*}{\alpha_0^*} \;\; \cdots - \frac{\alpha_{u-1}^*}{\alpha_0^*} \;\; -\frac{\alpha_u^*}{\alpha_0} \;\; -\frac{\alpha_{u+1}^*}{\alpha_0} \;\; \cdots - \frac{\alpha_{v-1}^*}{\alpha_0} \right]^T .$$

Notice that there are $v+1$ variables in Q_1, and only v equations in Eq. (3.1). The normalization equation is needed to solve Eq. (3.1) for Q_1:

$$q_0 + q_1 + \cdots + q_v = 1 - \xi, \text{ where } \xi = \sum_{k=v+1}^{k_{max}} q_k . \qquad (3.2)$$

Since ξ is usually very small ($< 10^{-4}$), Eq. (3.1) will give a good approximate solution even if ξ is set to zero.

We now want to compute $Q_1 = A_1^{-1} C$. Since A_1 is a lower triangular matrix, A_1^{-1} will also be a lower triangular matrix. For convenience, let $b_{i,j}$ and $c_{i,j}$ be the nonzero elements of A_1^{-1} and A_1, respectively, and let C_i be the elements of C. Then for $i, j = 1, \ldots, v$

$$\sum_{k=1}^{v} b_{i,k}\, c_{k,j} = \begin{cases} 1 & i = j \\ 0 & i \neq j. \end{cases}$$

Solving Eqs. (3.1) and (3.2) for $b_{i,j}$, we get:

$$\begin{aligned}
b_{i,j} &= 0, \quad j > i \\
b_{i,i} &= 1, \quad 1 \le i \le v \\
b_{i,i-1} &= -c_{i,i-1} = \frac{1-\alpha_1^*}{\alpha_0^*}, \quad 1 \le i \le u \\
b_{i,i-1} &= -c_{i,i-1} = \frac{1-\alpha_1^*}{\alpha_0}, \quad u+1 \le i \le v \\
b_{i,j} &= \frac{1-\alpha_1^*}{\alpha_0^*} - \sum_{k=2}^{i-j} \frac{\alpha_k^*}{\alpha_0^*}\, b_{i,j+k}, \quad 1 \le i \le u,\ 1 \le j \le i-1 \\
b_{i,j} &= \frac{1-\alpha_1^*}{\alpha_0^*} - \sum_{k=2}^{i-j} \frac{\alpha_k^*}{\alpha_0}\, b_{i,j+k}\,, \qquad u+1 \le i \le v,\ 1 \le j \le i-1.
\end{aligned}$$

Substituting $b_{i,j}$ into Eq. (3.1), we have

$$q_k = \left[\sum_{j=1}^{k} b_{k,j}\, C_j\right] q_0 \quad 1 \le k \le v. \tag{3.3}$$

Substituting q_k's in Eq. (3.3) into Eq. (3.2),

$$q_0 = \frac{1-\xi}{1 + \left[\sum_{k=1}^{v}\sum_{j=1}^{k} b_{k,j}\, C_j\right]}. \tag{3.4}$$

Then, q_k's for $1 \le k \le v$ can be determined by substituting Eq. (3.4) into Eq. (3.3).

For example, when $u = 1$, $f = 2$, $v = 3$, the distribution of queue length becomes:

$$\begin{aligned}
q_0 &= \frac{(1-\xi)\,\alpha_0^*\,\alpha_2^*}{\alpha_0^* - \alpha_2^* + (1+\alpha_0-\alpha_1)\,(1-\alpha_0^*-\alpha_1^*)\alpha_2^*} \\
q_1 &= \frac{1-\alpha_0^*}{\alpha_0^*}\, q_0 = \frac{(1-\delta)\,(1-\alpha_0^*)\alpha_2^*}{\alpha_0^* - \alpha_2^* + (1+\alpha_0-\alpha_1)\,(1-\alpha_0^*-\alpha_1^*)\alpha_2^*} \\
q_2 &= \frac{1-\alpha_0^*-\alpha_1^*}{\alpha_0\,\alpha_0^*}\, q_0 = \frac{(1-\xi)\,(1-\alpha_0^*-\alpha_1^*)\alpha_2^*}{\alpha_0\,(\alpha_0^* - \alpha_2^* + (1+\alpha_0-\alpha_1)\,(1-\alpha_0^*-\alpha_1^*)\alpha_2^*)} \\
q_3 &= \frac{1}{\alpha_0\,\alpha_0^*}\left[\frac{(1-\alpha_1)\,(1-\alpha_0^*-\alpha_1^*)}{\alpha_0} - \alpha_2^*\right] q_0.
\end{aligned}$$

3.2.2 Solving q_k for $k > v$

We now want to derive q_k's for $k > v$. The last formula in Eq. (2.5) was:

$$\begin{aligned} q_k &= \sum_{i=0}^{n} \varepsilon_i \, \alpha^*_{i+k} \, (q_0 + q_1) + \sum_{i=2}^{u} \left(\sum_{j=0}^{\infty} \varepsilon^*_j \, \alpha_{j+k-i+1} \right) q_i + \sum_{i=u+1}^{k+1} \left(\sum_{j=0}^{\infty} \varepsilon_j \, \alpha_{j+k-i+1} \right) q_i \\ &+ \sum_{i=k+2}^{\infty} \left(\sum_{j=0}^{\infty} \varepsilon_{j+i-k-1} \, \alpha_j \right) q_i \quad \text{for } k = v+1, \ldots \infty. \end{aligned}$$

To simplify this expression, define

$$\begin{aligned} \nu_i &\overset{def}{=} \sum_{j=0}^{\sigma} \varepsilon_j \, \alpha_{i+j}, \quad 0 \le i \le k_{max} \\ B_k &\overset{def}{=} \sum_{j=0}^{n} \varepsilon_j \, \alpha^*_{j+v+k} \, q_0 + \sum_{i=1}^{u} \left(\sum_{j=0}^{\infty} \varepsilon_j \, \alpha^*_{v+k+j-i+1} \right) q_i \\ &\quad + \sum_{i=u+1}^{v} \left(\sum_{j=0}^{\infty} \varepsilon_j \, \alpha_{v+k+j-i+1} \right) q_i \;, \quad 1 \le k \le k_{max}. \end{aligned}$$

Since $q_k \ll 1$ for $k > v$, the terms related to q_{k+2}, $k > v$, in Eq. (2.5) can be set to zero, yielding approximate equations:

$$\begin{array}{llll} (1-\nu_1)\, q_{v+1} & -\nu_0 \, q_{v+2} & & = B_1 \\ -\nu_2 \, q_{v+1} & +(1-\nu_1)\, q_{v+2} & -\nu_0 \, q_{v+3} & = B_2 \\ & \vdots & & \\ -\sum_{i=1}^{k-2} \nu_{k-i} \, q_{v+i} & + \, (1-\nu_1)\, q_{v+k-1} & -\nu_0 \, q_{v+k} & = B_{k-1} \\ 4 \le k \le k_{max}, & \text{where } q_{v+k_{max}} \ll P_{dyn}. & & \end{array}$$

The above equations can be rewritten as $A_2 Q_2 = B$, where $Q_2 = [q_{v+1} \; q_{v+2} \; \cdots \; q_{v+k_{max}} \;]^T$, $B = [B_1 \; B_2 \; \cdots \; B_{k_{max}-1} \;]^T$, and A_2 is a $(k_{max}-1) \times (k_{max}-1)$ lower triangular matrix if ν_0 is set to zero. The inverse of A_2 is calculated using the perturbation approach as shown below.

Calculation of A_2^{-1}

Let A_2^e be the original matrix and A_2 be the matrix after setting ν_0 to zero. Since A_2 is a lower triangular matrix, A_2^{-1} can be easily computed as follows. Let $a_{i,j}$ denote the nonzero $(i,j)^{th}$ element of A_2^{-1}, then

$$\begin{aligned} a_{i,i} &= \frac{1}{1-\nu_1} \\ a_{i,j} &= \sum_{k=1}^{i-j} \frac{\nu_{k+1}}{1-\nu_1} a_{i,j+k}, \quad 1 \le i \le k_{max},\ 1 \le j \le i-1. \end{aligned}$$

A few examples of $a_{i,j}$'s are:

$$\begin{aligned} a_{i,i} &= \frac{1}{1-\nu_1}, \quad 1 \le i \le k_{max} \\ a_{i,i-1} &= \frac{\nu_2}{(1-\nu_1)^2}, \quad 2 \le i \le k_{max} \\ a_{i,i-2} &= \frac{\nu_2^2}{(1-\nu_1)^3} + \frac{\nu_3}{(1-\nu_1)^2}, \quad 3 \le i \le k_{max}. \end{aligned}$$

Finally, the distribution of queue length can be calculated as:

$$q_{v+k} = \sum_{j=1}^{k} a_{k,j} B_j, \quad 1 \le k \le k_{max}. \tag{3.5}$$

Successive Approximation

The perturbation approach is applied to account for the effect of $\nu_0 \neq 0$. The basic idea of perturbation theory works as follows. Suppose the solution of a linear system $AX = B$ is known. When A and B change to $A + \epsilon A$ and $B + \epsilon B$, respectively, the new solution $X + \epsilon X$ can be derived from A and X instead of inverting $A + \epsilon A$ if ϵ is a small number [60].

Let Q_2^e and δA_2 be the exact distribution of queue length, and the matrix that contains ν_0, respectively. The solution to $A_2^e Q_2^e = B$ can be successively approximated as follows. Let $Q_2^e = Q_2 + \delta Q_2^e$. Since A_2^{-1} has already been calculated, Q_2 can be computed first. Then, $(A_2 - \delta A_2)\delta Q_2^e = \delta A_2 Q_2$, where δA_2 is a matrix of the same size as A_2 with ν_0 at element $(i, i+1)$, $1 \le i < k_{max}$, and zero otherwise. The same method can be used to solve for δQ_2^e. Let $\delta Q_2^e = \delta Q_2^{(1)} + \delta Q_2^r$, then $(A_2 - \delta A_2)(\delta Q_2^{(1)} + \delta Q_2^r) = \delta A_2 Q_2$. This equation can be divided into two parts which are then solved separately. The first part is $A_2 \delta Q_2^{(1)} = \delta A_2 Q_2$ and the second part is $(A_2 - \delta A_2)\delta Q_2^r = \delta A_2 Q_2^{(1)}$. Since we have already found A_2^{-1}, $\delta Q_2^{(1)}$ can be calculated by $\delta Q_2^{(1)} = A_2^{-1}\delta A_2 Q_2$. This method is applied again to solve for δQ_2^r in the second part. Let $\delta Q_2^r = \delta Q_2^{(2)} + \delta Q_2^{r'}$, then there are two equations similar to the previous iteration:

$$\begin{aligned} A_2\,\delta Q_2^{(2)} &= \delta A_2 Q_2^{(1)} \\ (A_2 - \delta A_2)\,\delta Q_2^{r'} &= \delta A_2 Q_2^{(2)}. \end{aligned}$$

Again, $\delta Q_2^{(2)}$ is calculated first $(= A_2^{-1}\ \delta A_2 Q_2^{(1)})$, and $\delta Q_2^{r'}$ is decomposed into $\delta Q_2^{(3)}$ and $\delta Q_2^{r''}$. Eventually, δQ_2^e can be approximated as the sum of an infinite series of $\delta Q_2^{(k)}$, $k = 1, \ldots, \infty$ as follows:

$$\begin{aligned} \delta Q_2^{(1)} &= A_2^{-1}\delta A_2 Q_2 \\ \delta Q_2^{(2)} &= A_2^{-1}\delta A_2 Q_2^{(1)} = (A_2^{-1}\delta A_2)^2 Q_2 \\ &\vdots \\ \delta Q_2^{(k)} &= A_2^{-1}\delta A_2 Q_2^{(k-1)} = (A_2^{-1}\delta A_2)^k Q_2 \end{aligned}$$

$$\delta Q_2^e = \delta Q_2^{(1)} + \delta Q_2^{(2)} + \cdots = \frac{(A_2^{-1}\delta A_2)}{I - A_2^{-1}\delta A_2} Q_2$$

$$\begin{aligned} Q_2^e &= Q_2 + \delta Q_2^e = Q_2 + \frac{(A_2^{-1}\delta A_2)}{I - A_2^{-1}\delta A_2} Q_2 \\ &= \frac{Q_2}{I - A_2^{-1}\delta A_2} = (I - A_2^{-1}\delta A_2)^{-1} Q_2. \end{aligned} \tag{3.6}$$

In most calculations, Q_2^e converges to a fixed constant after two to three iterations, it is not necessary to invert $(I - A_2^{-1}\delta A_2)$ in Eq. (3.6). Combining Eqs. (3.3) to (3.6), the distribution of queue length can be derived for any threshold patterns.

3.2.3 Deriving the Combined Task Arrival Rate ω

Two unknown parameters, ω and ϵ's in Eqs. (3.3) to (3.6) need to be derived before solving these equations. Deriving the combined task arrival rate ω is discussed in this section. In the proposed LSMSCB, a node is overloaded when its queue length is greater than v. Only overloaded node can transfer tasks to other underloaded nodes in its buddy set. The task transfer out rate (β) is shown as:

$$\begin{aligned}\beta &\equiv \left[\sum_{k=v+1}^{\infty} (k-v)\,\alpha_k^*\right](q_0+q_1) + \sum_{i=2}^{v+1}\left[\sum_{k=v+2-i}^{\infty} (k-v-1+i)\,\alpha_k\right] q_i \\ &+ \sum_{i=v+2}^{\infty}\left[\sum_{k=0}^{\infty} k\,\alpha_k\right] q_i. \end{aligned} \tag{3.7}$$

Note that β is the rate of task transfer out of a node. If all nodes' external task arrival rates are identical, then $\tau = \beta$. Otherwise, τ must be calculated by Eq. (2.10).

Combining τ and λ, the distribution of q_k's for $k \leq v$ can be calculated by Eqs. (3.3) and (3.4). Substituting the newly calculated q_k's into Eq. (3.7), one can get a new τ. The same procedure is applied again to adjust q_k's. This procedure will repeat until q_k's and ω converge to a fixed number.

3.2.4 Approximation Accuracy

Due to the complexity of Eq. (3.1), the closed-form solution is derived by separating it into two parts and solving them approximately. It is desirable to analyze the accuracy of the derived solution.

The task transfer-in rate τ is approximated by Eq. (3.7). Since ϵ cannot be determined before deriving τ, only q_k's for $k < v$ are used in Eq. (3.7). Let $\delta\tau$ be the difference between the derived results and the actual value, then

$$\begin{aligned}\delta\tau &= \left[\sum_{k=v+1}^{\infty} (k-v)\,\alpha_k^*\right](\delta q_0+\delta q_1) + \sum_{i=2}^{v+1}\left[\sum_{k=v+2-i}^{\infty} (k-v-1+i)\,\alpha_k\right] \delta q_i + \\ &\quad \sum_{i=v+2}^{\infty}\left[\sum_{k=0}^{\infty} k\,\alpha_k\right] \delta q_i \end{aligned}$$

where

$$\delta q_0 = \frac{-\xi}{1+\left[\sum_{k=1}^{v}\sum_{j=1}^{k} b_{k,j}\,C_j\right]}$$

$$\begin{aligned} \delta q_k &= \left[\sum_{j=1}^{k} b_{k,j}\, C_j\right] \delta q_0 \quad 1 \le k \le v \\ \delta q_k &= q_k \quad k > v. \end{aligned} \tag{3.8}$$

The variation of τ will in turn affect the distribution of queue length. As an example, the changes of τ and queue length are studied for the threshold pattern $u = 1$, $f = 2$, $v = 3$ with buddy set size 10 and system load $\rho = 0.8$. As shown in Table 3.1, the transfer-in task rate is 0.059 (0.065) when ξ is omitted (considered); the difference is about 10%. The change of q_k's is around 1% due to the variation of τ. The beauty of the proposed approximate method in deriving τ is that q_k's for $k \le v$ are increased a little by omitting ξ while q_k's for $k > v$ are completely ignored to compensate for the omission of ξ. So, τ can be derived quite accurately.

The variation of ϵ_ks vs. the change of distribution of queue length is given in Table 3.2. The variation of ϵ_ks is found significantly larger than the change of q_ks for $k \le 3$. The variation of q_k's for $k > 3$ is found to be about 10% due to the change of ϵ_k's. Note that the variation of q_k's for $k > 3$ given in Table 3.1 is derived by only considering the variation of τ (ϵ_k's are assumed unchanged). It is clear that ϵ_k's dominate the variation of q_ks for $k > 3$.

Summarizing the above analysis, we found that the first part of q_k's (for $k \le v$) is not sensitive to the second part of q_k's (for $k > v$) as long as ξ is small. However, the second part of q_k's is very sensitive to the accuracy of the first part of q_k's. Since the approximate closed-form solutions determine the first part of q_k's very accurately, the inaccuracy of the second part of q_k's and P_{dyn} is within 10% of the corresponding true value when $\xi \le 10^{-4}$.

3.3 Design of an Optimal LSMSCB

The QL derived from Eqs. (3.3) to (3.6) is verified/compared with the results derived in Section 2.5. As shown in Fig. 3.1, in most cases, the QL derived from the closed-form equations is closer to the true (simulation) result and is also consistent with the results calculated numerically. The QL derived from the numerical method is always smaller than the closed-form result, because some of the high-order coefficients were ignored in the numerical method due to the use of restricted matrix size.

Since P_{dyn} depends on the system utilization and task deadlines, there are upper bounds of system utilization and deadlines for any given value of P_{dyn}, denoted by P^*_{dyn}. For example, as shown in Fig. 3.2, both threshold patterns '1 2 3' and '2 4 5' failed to meet the specification if $P^*_{dyn} = 10^{-8}$ and deadline $D < 5$ with system load higher than 0.5. On the other hand, to ensure $P_{dyn} \leq P^*_{dyn}$ even with system load = 0.9, the task deadlines must be greater than 12. Due to the nature of LSMSCB, minimizing P_{dyn} will increase the communication overhead. The communication cost, denoted as C_{com}, includes the overheads associated with task transfers and state-information collection. The cost of collecting state information is determined by the product of the frequency of state change (f_{sc}) and the size of buddy set (σ). The task transfer cost (β) can be controlled by adjusting TH_v; a higher TH_v will lower β. For example, minimizing C_{com} can be achieved by reducing the frequency of state change, the buddy set size, or the task transfer rate. The frequency of state change can be reduced by increasing the difference between TH_u and TH_f. A small buddy set size will reduce the cost for collecting state information, but will increase P_{dyn}. The task transfer rate can be reduced by increasing TH_v. However, P_{dyn} will generally increase with any attempt to reduce C_{com}. Thus, the primary goal in the design of an optimal LSMSCB is either to ensure the system's P_{dyn} to be lower than P^*_{dyn} while minimizing the communication cost, or to minimize P_{dyn} while keeping the communication cost below a pre-specified level, C^*_{com}.

The problem of optimizing LSMSCB is formally stated as follows.

Optimal LSMSCB1: Minimize $C_{com} = \sigma f_{sc} + \beta$ subject to $P_{dyn} \leq P^*_{dyn}$.

Optimal LSMSCB2: Minimize P_{dyn} subject to $C_{com} \leq C^*_{com}$.

From the analysis in Section 2.4, f_{sc} is equal to the total number, T, of arriving tasks times the probability, P_{sc}, of state change. P_{dyn} can be computed as $\sum_{k=D}^{k_{max}} q_k$, where D is the given task deadline. The closed-form equations can be used to derive the optimal threshold patterns and buddy set sizes for both the minimization problems.

Since P_{dyn} and C_{com} are determined by the threshold pattern and buddy set size used, the optimal threshold pattern and buddy set size can be found by an exhaustive search. In such a case, the search complexity is $O(D^3 N)$, where N is the total number of nodes and D is the given deadline. This complexity can be greatly reduced by utilizing the results in [25] as follows. First, the frequency of state change is 100% if $TH_f = TH_u$, and f_{sc} can be greatly reduced if $TH_f - TH_u \geq 1$, or $TH_f \geq TH_u + 1$. Second, the number of unnecessarily transferred tasks will increase if TH_f is close to TH_v, thus increasing the probability of missing task deadlines. This effect can be explained by the following example. Let $P_{dyn}^{(1)}$ and $P_{dyn}^{(2)}$ denote the probability of missing task deadlines under two threshold patterns with the same TH_u and TH_v, but $TH_f^{(1)}$ and $TH_f^{(2)}$, respectively. If $TH_f^{(1)} < TH_f^{(2)} < TH_v$ and $P_{dyn}^{(1)} < P_{dyn}^{(2)}$, then it is impossible to find a threshold pattern with the same TH_u and TH_v, but a TH_f greater than $TH_f^{(2)}$ that results in a lower P_{dyn} than $P_{dyn}^{(1)}$. In other words, the search for an optimal solution can skip all of the threshold patterns with such a TH_f.

Third, the effect of changing buddy set size on P_{dyn} is quite complicated due to the interaction between nodes in a buddy set. As shown in Fig. 3.3, P_{dyn} continues to decrease with the increase of buddy set size when system load is 0.5, and P_{dyn} approaches a constant when system load is 0.7, but it may even increase with the increase of buddy set size when system load is higher than 0.9. Nonetheless, buddy sets of larger than 20 nodes will only reduce P_{dyn} infinitesimally, and cannot offset the increasing cost of state-change broadcasts.

Based on the above observations, one can find the optimal threshold pattern and optimal buddy set size subject to the given D, P^*_{dyn}, and C^*_{com} by the following procedure. The first phase is to find a threshold pattern that satisfies the constraint. The search starts at $TH_u = 0$, $TH_f = TH_u + 1$, $TH_v = TH_f + 1$, and $\sigma = 20$ (or any other large number). If the resulting $P_{dyn} > P^*_{dyn}$, increase TH_v until it becomes equal to D. Then, increment TH_f or TH_u, and restart the search until a pattern that satisfies the constraint is found. The next phase is either to minimize C_{com} for the optimal LSMSCB1, or to minimize P_{dyn} for the optimal LSMSCB2. The former is done by increasing the difference between TH_u and TH_f, or increasing TH_v, or reducing σ. Since any of these attempts will increase P_{dyn}, once the constraint cannot be satisfied, the minimizing procedure should stop. The optimal LSMSCB2 is solved by increasing TH_u and buddy set sizes until the constraint cannot be satisfied.

3.3.1 Optimization results

Many important and useful results are drawn from the above optimization process. First, the minimum achievable P_{dyn} with no constraint on C_{com} is given in Fig. 3.4. The results show that P_{dyn} can be reduced to below 10^{-15} when $D > 7$ and $\lambda = 0.8$. In most cases, P_{dyn} can be easily reduced to below 10^{-7}. The next figure shows the minimum achievable P_{dyn} subject to the constraint $C_{com} \leq C^*_{com}$. The cost shown in Fig. 3.5 is normalized to the total number of arrived tasks on a node, so $C_{com} = 1.0$ means that one control message will be generated per task arrival. Fig. 3.6. shows the minimum achievable C_{com} while keeping $P_{dyn} \leq P^*_{dyn}$. When $\lambda = 0.5$ and $D > 6$, LSMSCB can reduce C_{com} to below 1% while keeping $P_{dyn} \leq 10^{-6}$. Even when $\lambda = 0.8$ and $P^*_{dyn} = 10^{-10}$, C_{com} can still be reduced to below 0.1 if $D > 10$.

Another important result found in the derived optimal threshold pattern and buddy set is that the system utilization is significantly increased, as compared to the results calculated based on the upper bound model in Section 2.5. For example, as shown in

Fig. 3.7, the external task arrival rate is increased from 0.3, 0.06, 0.03 (from Fig. 2.9) to 0.88, 0.7, 0.57 when $D = 4$ and $P_{dyn} = 10^{-4}, 10^{-6}, 10^{-8}$, respectively.

Some optimal solutions found for the various values of D and P^*_{dyn} are given in Table 3.3. This table is useful in the design of real–time systems for the following reasons. First, the system can be easily verified if it can meet the specification. The empty entries indicate the nonexistence of such a system. Second, a solution can always be found under any given D and P^*_{dyn} if such a solution exists. Third, an existing solution is optimal in the sense that the system constraint is satisfied and the communication cost is minimized.

	$\xi = 0$	$\xi = 4.9 \times 10^{-4}$	Variation	Variation in %
β	0.59	0.65	0.06	10.17
q_0	0.2352	0.2327	-0.0025	-1.08
q_1	0.3203	0.3200	-0.0003	-0.1
q_2	0.2630	0.2641	0.0011	0.44
q_3	0.1816	0.1828	0.0012	0.66
q_4	4.03×10^{-4}	4.06×10^{-4}	3.08×10^{-6}	0.76
q_5	7.26×10^{-5}	7.30×10^{-5}	5.6×10^{-7}	0.77
q_6	1.10×10^{-5}	1.12×10^{-6}	8.72×10^{-8}	0.78
q_7	1.48×10^{-6}	1.49×10^{-6}	1.17×10^{-8}	0.79
q_8	1.70×10^{-7}	1.72×10^{-7}	1.4×10^{-9}	0.82

Table 3.1: Variation of q_k's and τ by omitting ξ.

	$\xi = 0$	$\xi = 4.9 \times 10^{-4}$	Variation	Variation in %
ϵ_0	0.001749	0.001932	0.000183	10.46
ϵ_1	0.015509	0.016771	0.001262	8.14
ϵ_2	0.061889	0.065514	0.003625	5.86
ϵ_3	0.146354	0.151663	0.005309	3.63
q_4	4.03×10^{-4}	4.35×10^{-4}	3.2×10^{-5}	7.9
q_5	7.26×10^{-5}	7.87×10^{-5}	6.1×10^{-6}	8.4
q_6	1.10×10^{-5}	1.21×10^{-5}	1.1×10^{-6}	10.0
q_7	1.48×10^{-6}	1.62×10^{-6}	1.4×10^{-7}	9.46
q_8	1.70×10^{-7}	1.87×10^{-7}	1.7×10^{-8}	10.0

Table 3.2: Variation of q_k's vs. the change of ϵ_k.

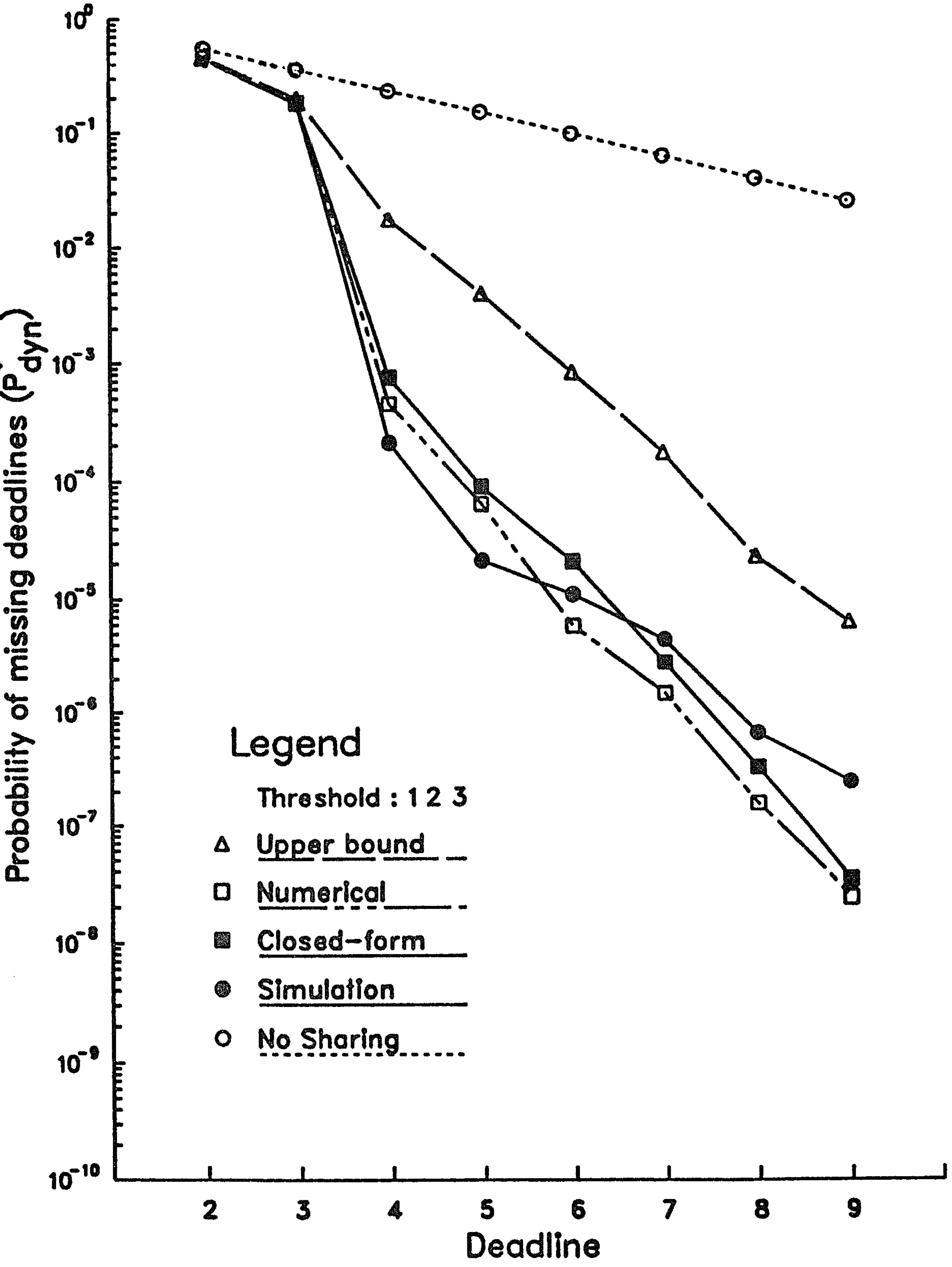


Figure 3.1: Comparison of P_{dyn} derived from different solution methods.

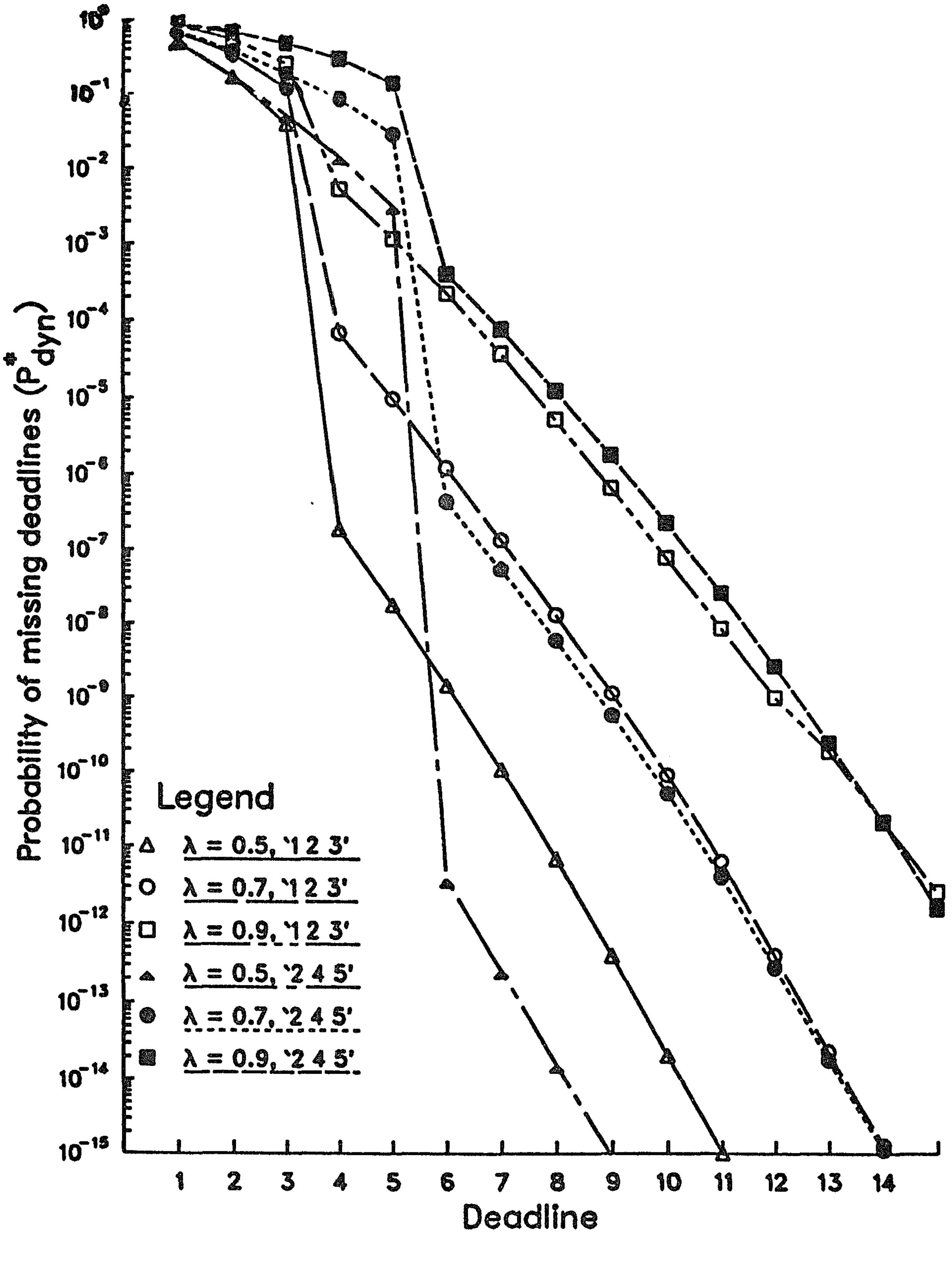


Figure 3.2: P_{dyn} vs. deadlines in different system loads and threshold patterns.

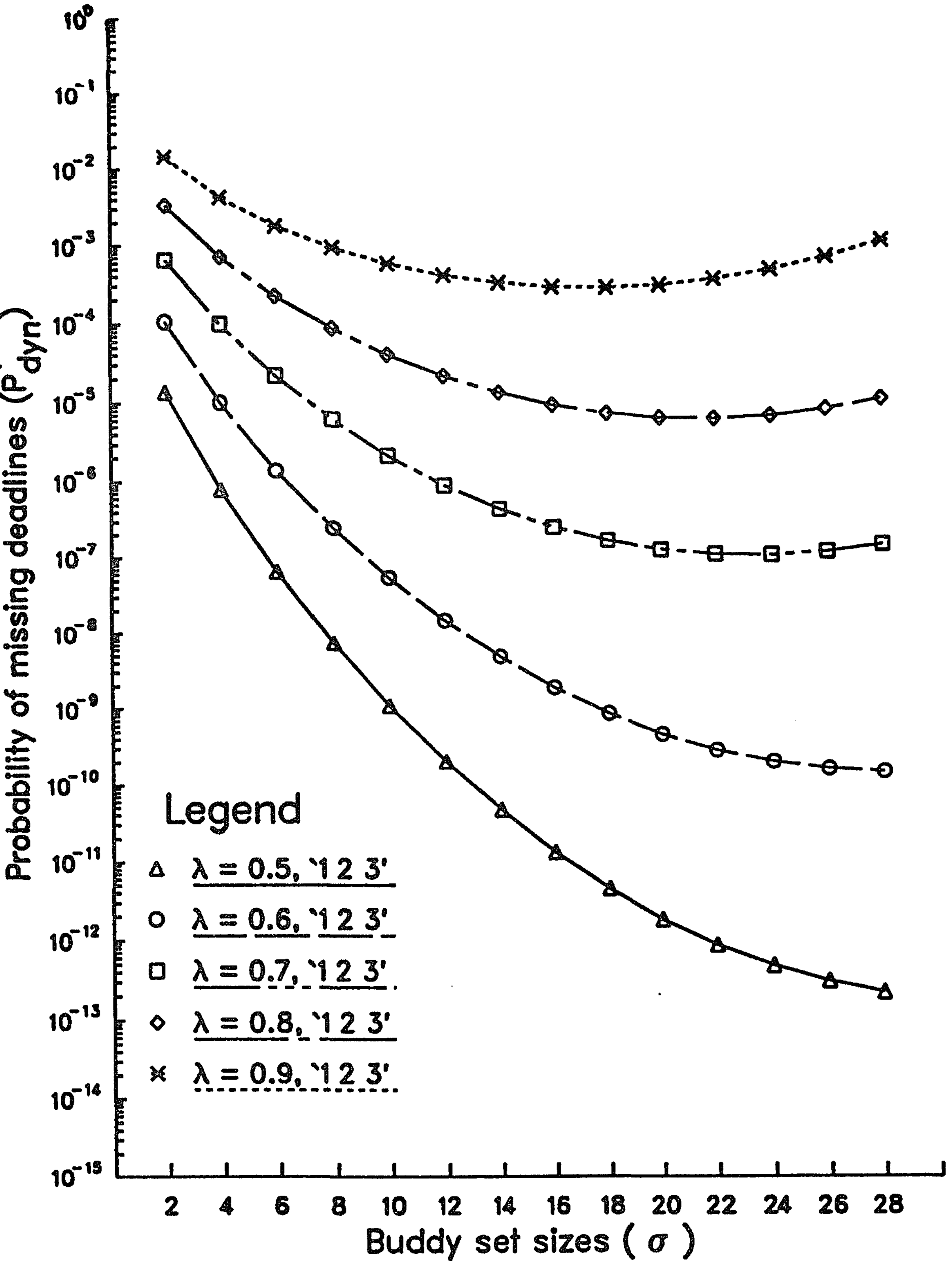


Figure 3.3: P_{dyn} vs. buddy set sizes ($D = 6$).

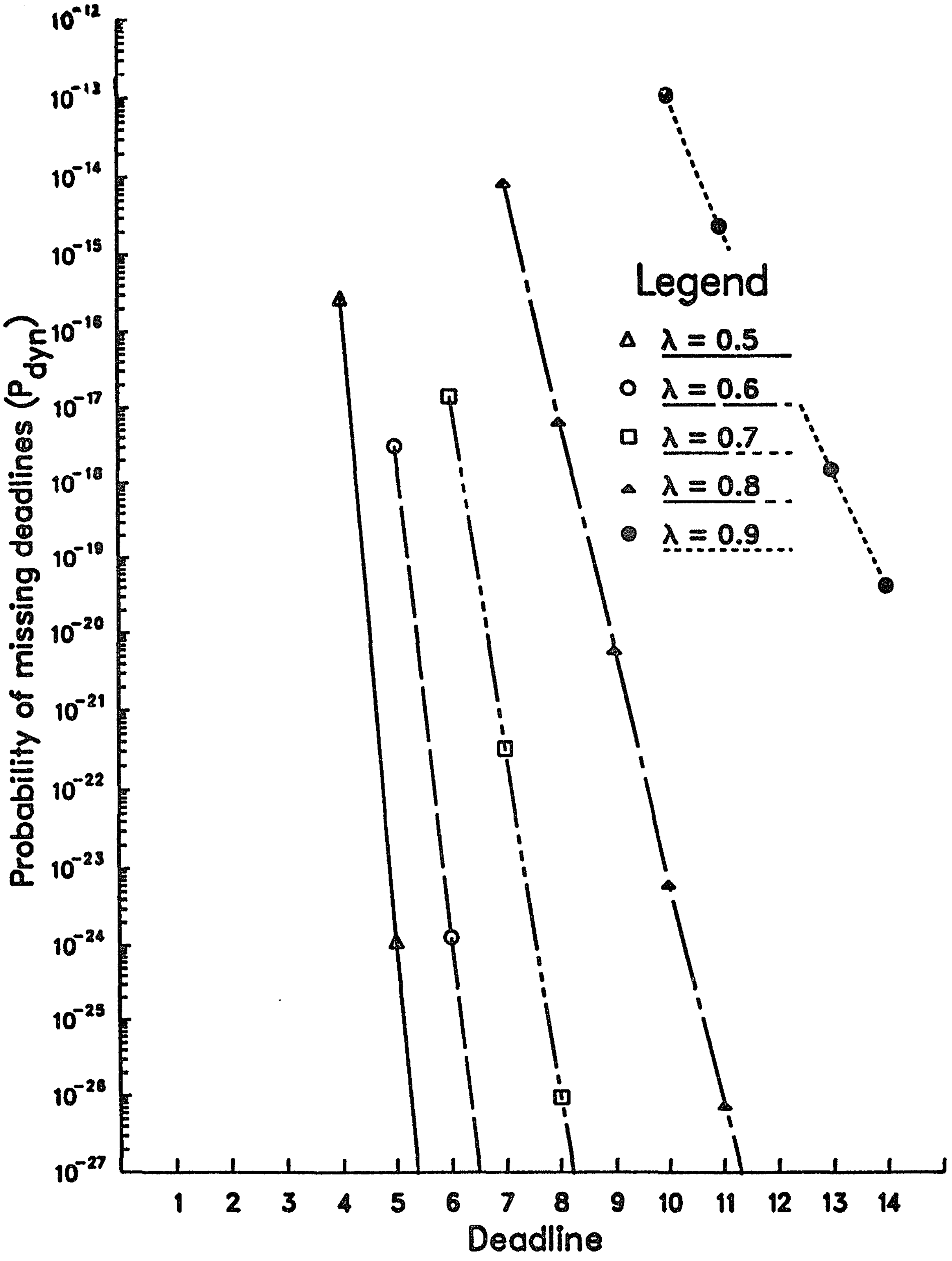


Figure 3.4: Minimal P_{dyn} under no constraint on communicatoin cost.

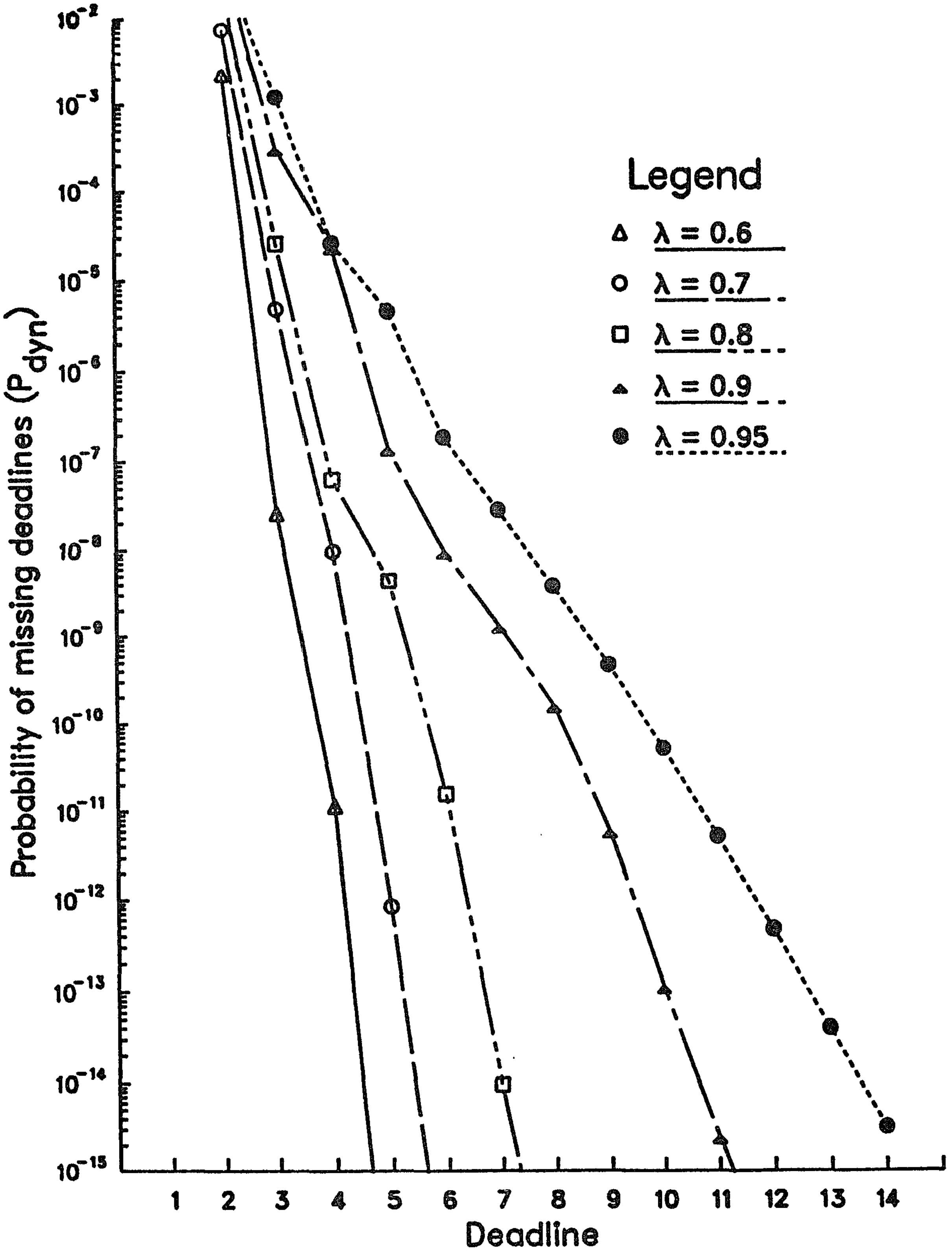


Figure 3.5: Minimal P_{dyn} when $C_{com} \leq 1.0$.

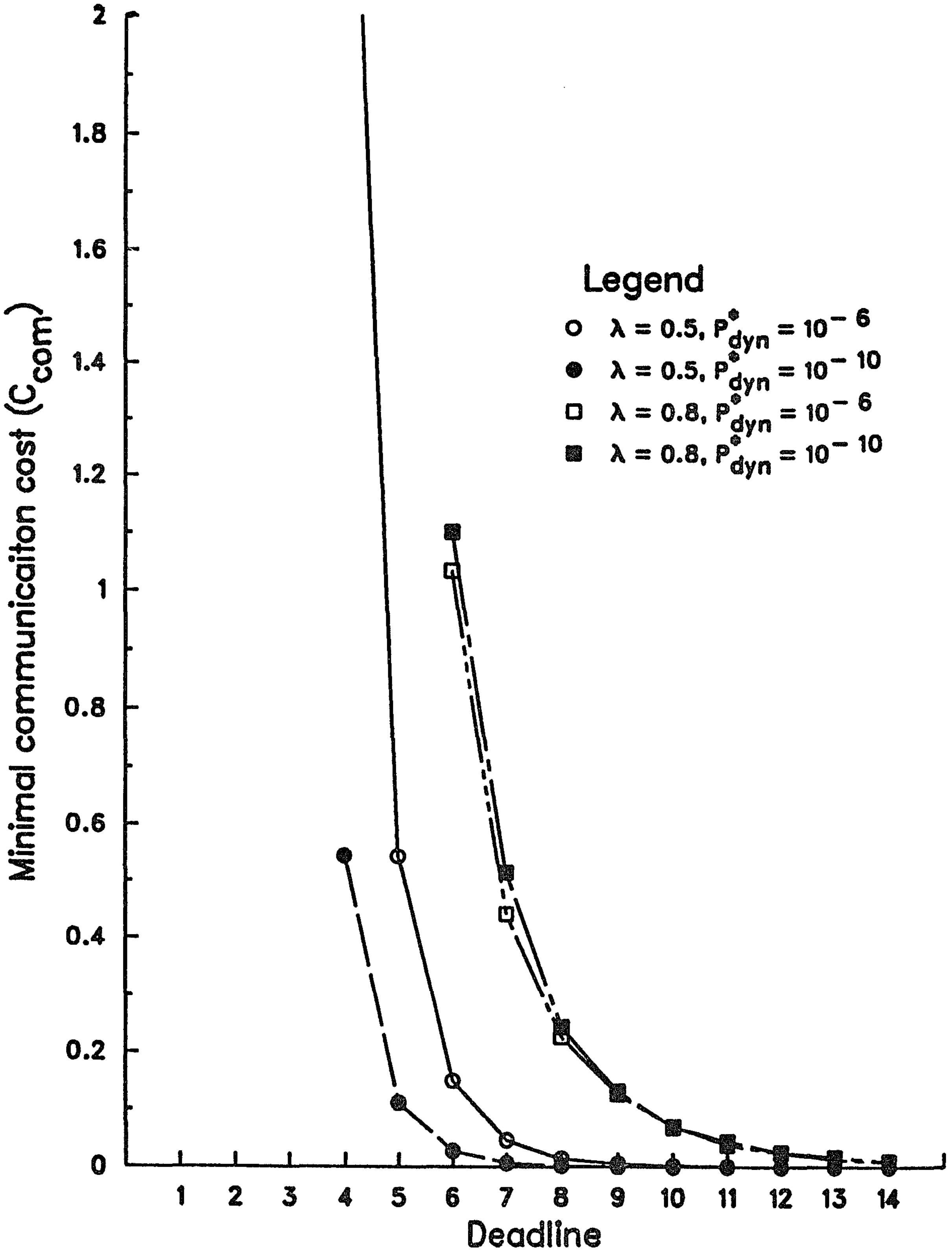


Figure 3.6: Minimal C_{com} under different system loads and P^{*}_{dyn}.

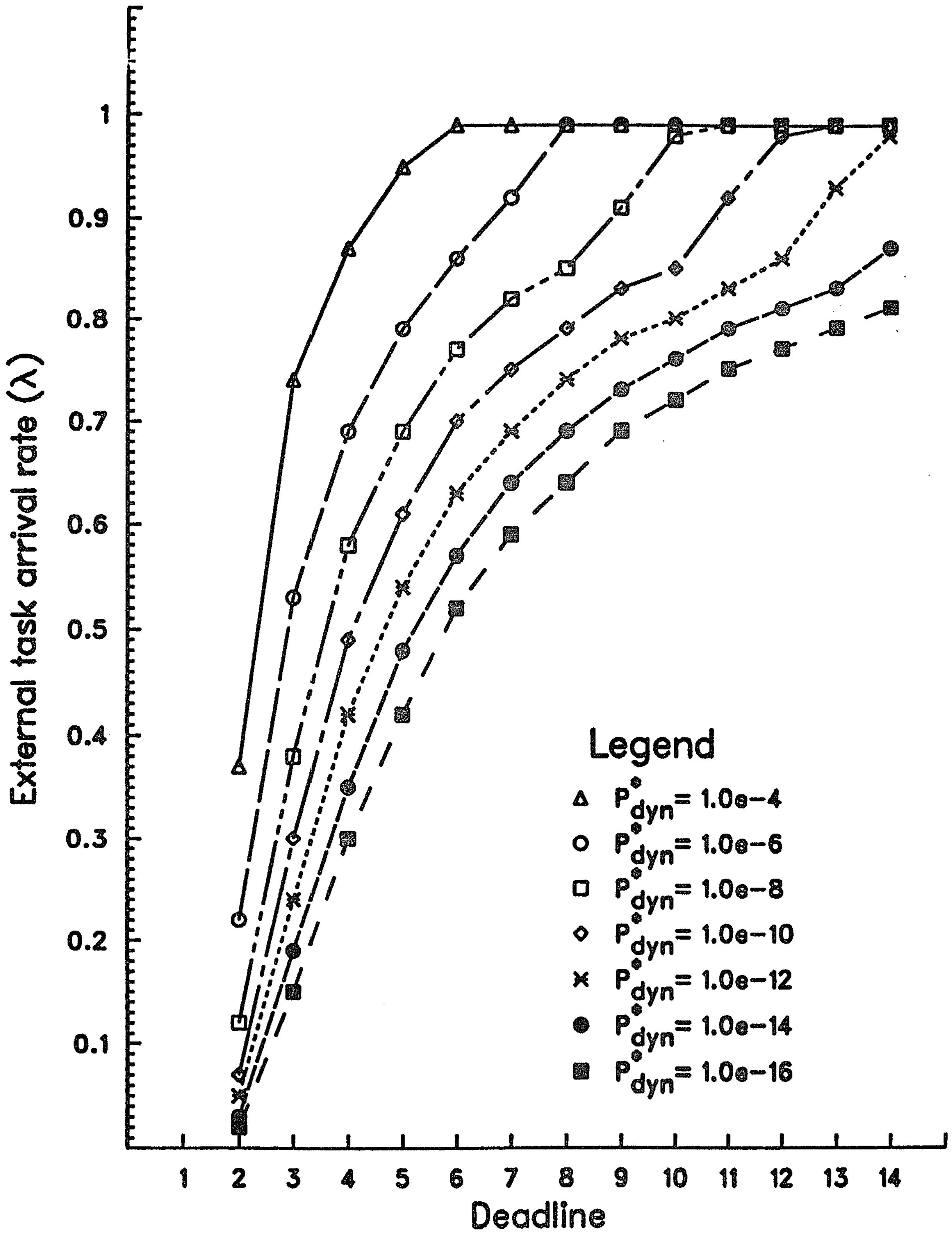


Figure 3.7: System utilization vs. deadlines

	Load (λ)	0.5	0.7	0.9
P^*_{dyn}	Deadline	TH_u TH_f TH_v σ	TH_u TH_f TH_v σ	TH_u TH_f TH_v σ
10^{-6}	≤ 2	—	—	—
	3	0 1 2, 10	—	—
	4	1 2 3, 5	2 3 4, 9	—
	5	1 2 3, 5	2 3 4, 7	2 3 4, 15
	6	2 4 5, 3	2 4 5, 6	3 4 5, 13
	7	2 5 6, 3	3 5 6, 6	3 5 6, 13
	8	4 6 7, 2	4 6 7, 5	4 5 6, 11
10^{-8}	≤ 3	—	—	—
	4	1 2 3, 9	1 2 3, 14	—
	5	1 3 4, 8	2 3 4, 12	—
	6	2 4 5, 6	2 4 5, 9	3 4 5, 17
	7	3 5 6, 5	4 5 6, 7	3 4 5, 14
	8	4 6 7, 5	4 6 7, 7	5 6 7, 14
	9	5 7 8, 4	5 7 8, 6	5 7 8, 13
10^{-10}	≤ 3	—	—	—
	4	1 2 3, 13	—	—
	5	1 3 4, 8	2 3 4, 16	—
	6	2 4 5, 7	2 4 5, 10	—
	7	4 5 6, 6	4 5 6, 9	—
	8	4 6 7, 6	5 6 7, 8	—
	9	5 7 8, 5	5 7 8, 8	6 7 8, 16

Table 3.3: Optimal threshold patterns and buddy set sizes under given D and P^*_{dyn}.

CHAPTER 4

A COORDINATED LOCATION POLICY IN HYPERCUBE MULTICOMPUTERS

4.1 Introduction

A variety of topologies have been proposed to interconnect the processors in distributed systems. Among these topologies, the hypercube has drawn considerable attention due mainly to its regularity, potential for fault-tolerance, and scalability over a useful range [61, 62, 63, 64, 65]. Various research issues in hypercube multicomputers have been addressed, such as fault-tolerant routing and broadcasting [66, 67, 68, 69], and subcube allocation [70]. However, very little on workload distribution/redistribution to meet certain criteria (e.g., task deadlines) in hypercube multicomputers has been reported in the open literature. Bursty task arrivals may temporarily overload some hypercube nodes while leaving others underloaded. This chapter is concerned with the application of the LSMSCB to hypercube multicomputer systems.

Any efficient LS approach requires to resolve the issues of determining the load state of each node, collecting and updating state information, locating underloaded nodes and transferring overflow tasks [35, 40, 55]. Since the issues of determining the load state and collecting/updating state information have been treated in Chapter 2 and 3, we will in this Chapter focus on the location policy. Unless properly coordinated in locating underloaded nodes, overloaded nodes may 'dump' many overflow tasks on a few underloaded nodes, thus needing to transfer some of the transferred–in tasks again [71]. In such a case, LS may not improve system response time even at the expense of communication overheads resulting from transferring tasks and/or collecting load states of other nodes.

Unfortunately, almost all existing location policies have deficiencies of one form or another. In the probing method [40], an overloaded node selects a node (a candidate receiver) and checks whether or not this node can share its load. If it can, the node will transfer an overflow task to that node; otherwise, it will probe another node. The communication overhead for probing nodes was shown to depend only on the number of probes regardless of system size. However, probing nodes is inefficient, because not only will it introduce an additional delay in completing the tasks to be transferred, but also the overloaded node may not be able to locate a receiver within a pre-specified number of probes if there are only a few underloaded nodes in the system. In case a *bidding algorithm* [32, 33, 34] is used, an overloaded node first broadcasts a request for bids. Three to four rounds of messages have to be generated and exchanged before a task is actually transferred. For the purpose of load balancing, if an underloaded node's bid is accepted by more than one overloaded node, the underloaded node may be able to accept the task from only one of the overloaded nodes, and thus, the rest of the overloaded nodes must start another round of bidding. This bidding process usually causes excessive communication delays to the tasks to be transferred. To reduce the number of messages in the bidding algorithm, Ni and Xu proposed a *drafting algorithm* by allowing underloaded nodes to initiate the drafting process [35]. This algorithm is shown to be able to reduce the number of message exchanges to about one third of that of a bidding algorithm. However, if an underloaded node is accepted by more than one overloaded node, it may not be able to accept tasks from all overloaded nodes.

Any good location policy must be the result of resolving the following two issues. First, when the number of overloaded nodes is less than, or equal to, the number of underloaded nodes, no more than one overloaded node should be allowed to select the same underloaded node as a receiver. Simultaneous selection of an underloaded node by multiple overloaded nodes results from lack of coordination among them, and will henceforth be called the *coordination* problem [71]. Specifically, n *task collisions* are said to occur if $n+1$ overflow tasks are sent simultaneously to an underloaded node.[1] Second, to minimize the task transfer delay, the buddy set of a node is composed of those nodes in its physical proximity (e.g., those one or two hops away). The overloaded nodes in a buddy set should be able to transfer their tasks to the nodes in different buddy sets such that those tasks arriving at overloaded nodes within a *hot region* — a region where the number of overloaded nodes is greater than that of underloaded nodes — can be shared throughout the entire system, not just by these nodes in the same buddy set. Formation of hot region(s) with an excessive number of overflow tasks will henceforth be called the *congestion* problem.

The main purpose of this chapter is to solve both the coordination and congestion problems in hypercube multicomputers. Specifically, it will be formally proved that the method of constructing the preferred list discussed in Section 2.3 can minimize the probability that more than one overloaded node transferring overflow tasks to an underloaded node and distribute the overflow tasks in a buddy set over many different buddy sets, rather than overloading the nodes in its own buddy set. Based on modeling and simulation, the LS method with the proposed preferred lists will be shown to outperform the bidding or drafting algorithms and other LS methods that are either based on probing or without using preferred lists.

In Section 4.2, the differences between the LSMSCB and the drafting algorithm are discussed first. Then the problems of coordination and congestion which may be encountered when implementing the LSMSCB are discussed. These problems are resolved by properly constructing the preferred list of each node. Analysis and construction of preferred lists

[1]This is to reflect the severity of the coordination problem. For example, sending three tasks to a node would be more severe than sending two tasks to the node.

are the subject of Section 4.3. Using the number of task collisions as a yardstick, the performance of the LSMSCB with the proposed solutions to the coordination and congestion problems is evaluated in Section 4.4. In Section 4.5, the proposed location policy is compared with the random selection, probing, bidding, and drafting algorithms.

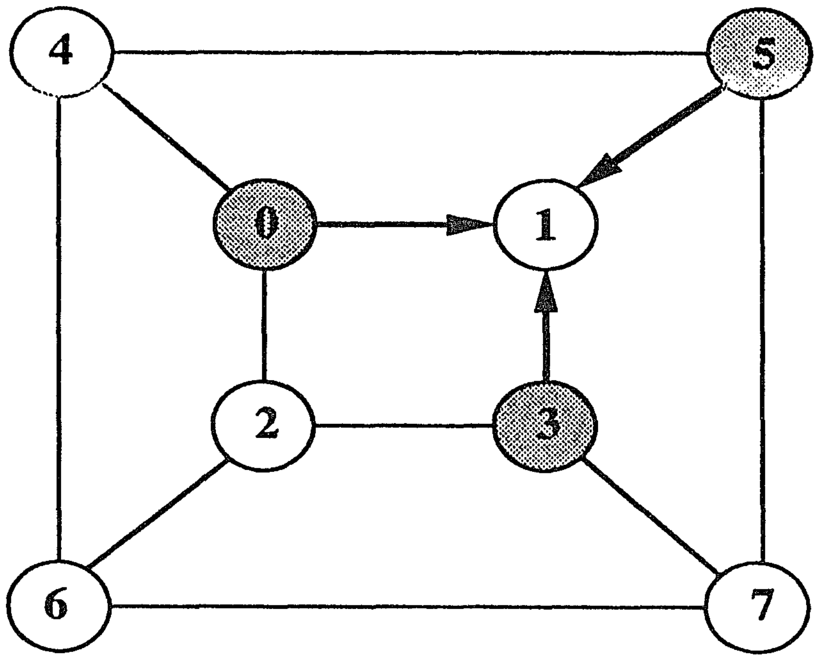
4.2 Advantages and Problems of the LSMSCB

As discussed in Section 2.3, three thresholds, TH_u, TH_f, and TH_v, are used to determine the state of a node. Since the main intent of this chapter is to solve the coordination and congestion problems in realizing the LSMSCB, it is first compared with the drafting algorithm [35] which is a similar, most commonly used method. The LSMSCB uses state–change multicasts within the buddy set of each node to collect and update the state information. This differs from other LS methods in the following aspects. Although the three thresholds defined in Section 2.3 are based on an idea similar to the three load states (L-, N-, and H- state) used in the drafting algorithm [35], their functions are quite different. For example, task transfer is initiated by an underloaded node in the drafting algorithm (i.e., receiver–initiated), and by an overloaded node in LSMSCB. The difficulty of a receiver–initiated method lies in that once a node switches to H-state, this state–change must be multicast to other nodes so that this 'new' overloaded node may be drafted by an underloaded node as soon as possible. A large number of control messages need to be multicast if the load switches frequently between N-state and H-state. Although many protocols were considered in [35], this problem could not be solved completely without causing drafting delays to H-state nodes. By contrast, since the LSMSCB is sender–initiated with receiver–initiated state updates, an overloaded node can locate, without any delay, an underloaded node using its preferred list. So, the state wiggling between TH_f and TH_v will not introduce any communication overhead at all, because the underloaded nodes need not know the exact state of overloaded nodes. Moreover, the state wiggling between TH_u

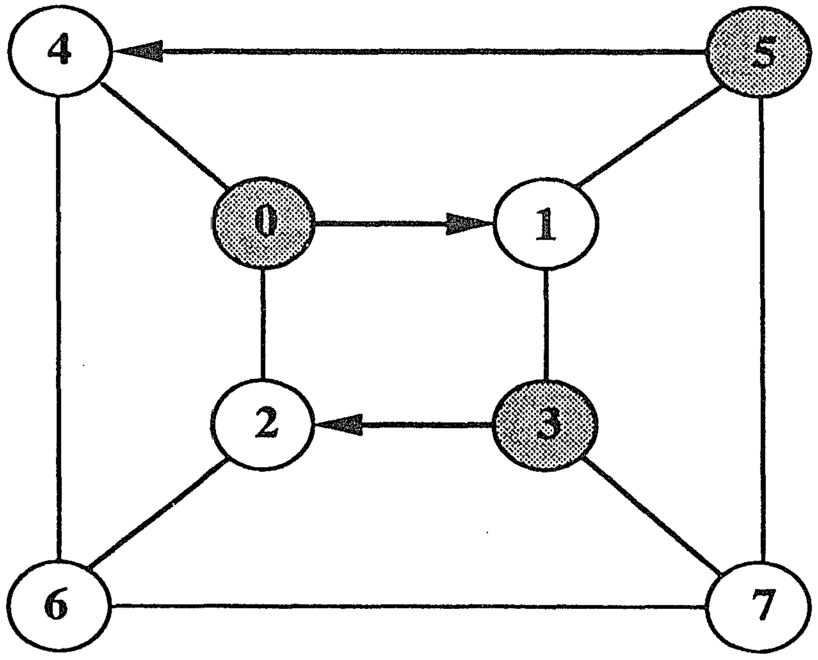
and TH_f can be reduced by increasing the magnitude of $TH_f - TH_u$ without affecting the system performance as long as there are some underloaded nodes.

Since the cost of task transfer increases with the physical distance between the sender and receiver nodes, the receiver node must be located as closely to the source node as possible. Thus, it is desirable to restrict a node's loads to be shared with those nodes in its proximity, i.e., *local* LS. Although a similar idea was adopted by others [35, 72, 55], they did not address several potential drawbacks of local LS. For example, if all the neighbors of an overloaded node are also overloaded while there are still many underloaded nodes elsewhere in the system, the conventional local LS will not be able to locate the underloaded nodes. Furthermore, the local LS in a hot region will tend to confine the overflow tasks only to a small number of nodes, thus making it impossible to balance/share the system load globally. By contrast, the buddy set used in the LSMSCB is constructed based on the distance between the sender and receiver nodes. The *first* entry of a node's preferred list consists of those nodes one hop away from the node, and the *second* entry consists of those nodes two hops away from the node, and so on. As was discussed in Section 2.5, the size of buddy set depends on system load and topology. For example, buddy sets of 10 to 15 nodes are shown to provide good performance over a wide range of system load. Note, for example, that the buddy set of a node in a 6-cube is formed with those nodes one and two hops away from the node.

However, both the coordination and congestion problems must be resolved before realizing the LSMSCB. These problems can, of course, be resolved easily if every overloaded node knows the receivers of other overloaded nodes. It is, however, practically impossible to equip every overloaded node with this knowledge, because each overloaded node is required to communicate with all the other overloaded nodes before transferring an overflow task to an underloaded node. This will not only introduce a great deal of communication traffic, but also result in severe delays in completing the tasks to be transferred, thus offsetting any



(a). Two collisions



(b). No collision

Figure 4.1: Coordination problem

benefit to be gained from LS.

One must therefore establish a rule for selecting a receiver among possibly multiple underloaded nodes while minimizing the probability of more than one overloaded node simultaneously transferring overflow tasks to the same underloaded node. For example, consider Fig. 4.1, where node '0', '3', and '5' are overloaded and all other nodes are underloaded. Unless properly coordinated, the three overloaded nodes may choose node '1' as the receiver, thus overloading node '1' and leaving the other nodes underloaded. Similarly, the congestion problem is illustrated in Fig. 4.2. Suppose nodes '0', '1', '2', '4', and '8' are in the same buddy set and three of them are overloaded (shaded circles), then they should transfer overflow tasks to the nodes outside of the buddy set. This problem can be resolved if the overloaded nodes select the underloaded nodes using preferred lists. As shown in Fig. 4.3, the overflow tasks within a hot region can be shared by the underloaded nodes outside of the region, thus spreading the overflow tasks more evenly over the entire system.

4.3 Construction of Preferred Lists and Buddy Sets

The congestion problem can be solved by designating each node in an n-cube as the k^{th} preferred node of one and only one other node. In such a case, since each node is designated as the most preferred node of one and only one node, the probability of an underloaded node being selected for task transfer by more than one overloaded node will be very small. Moreover, to enable the entire system to share the overflow tasks in any hot region, the most preferred receiver of each node in a buddy set must belong to a different buddy set. Since an overloaded node is most likely to transfer an overflow task to its most preferred receiver, the overloaded nodes in a buddy set will spread their overflow tasks over many different buddy sets, thus solving the congestion problem.

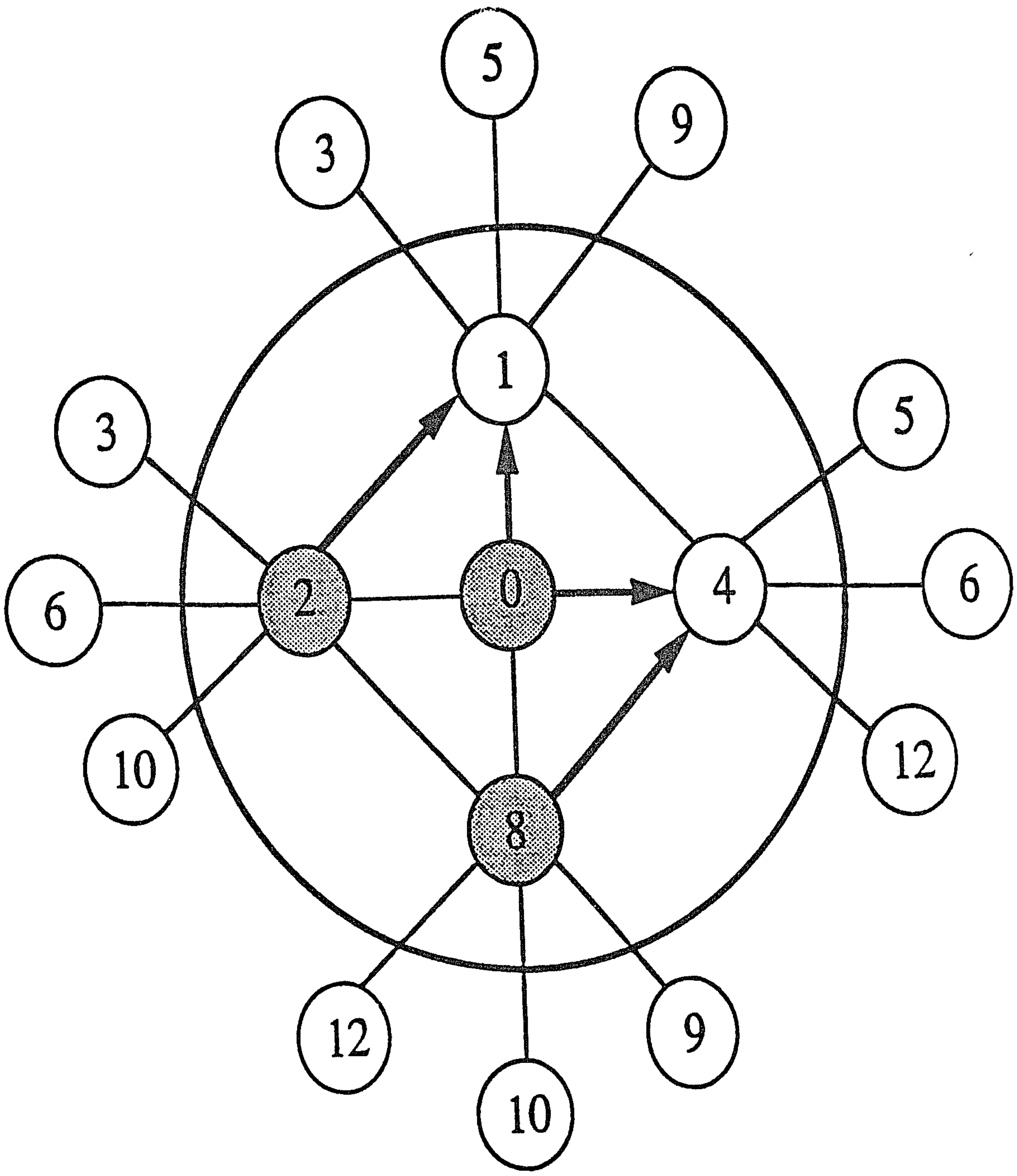


Figure 4.2: Congestion problem

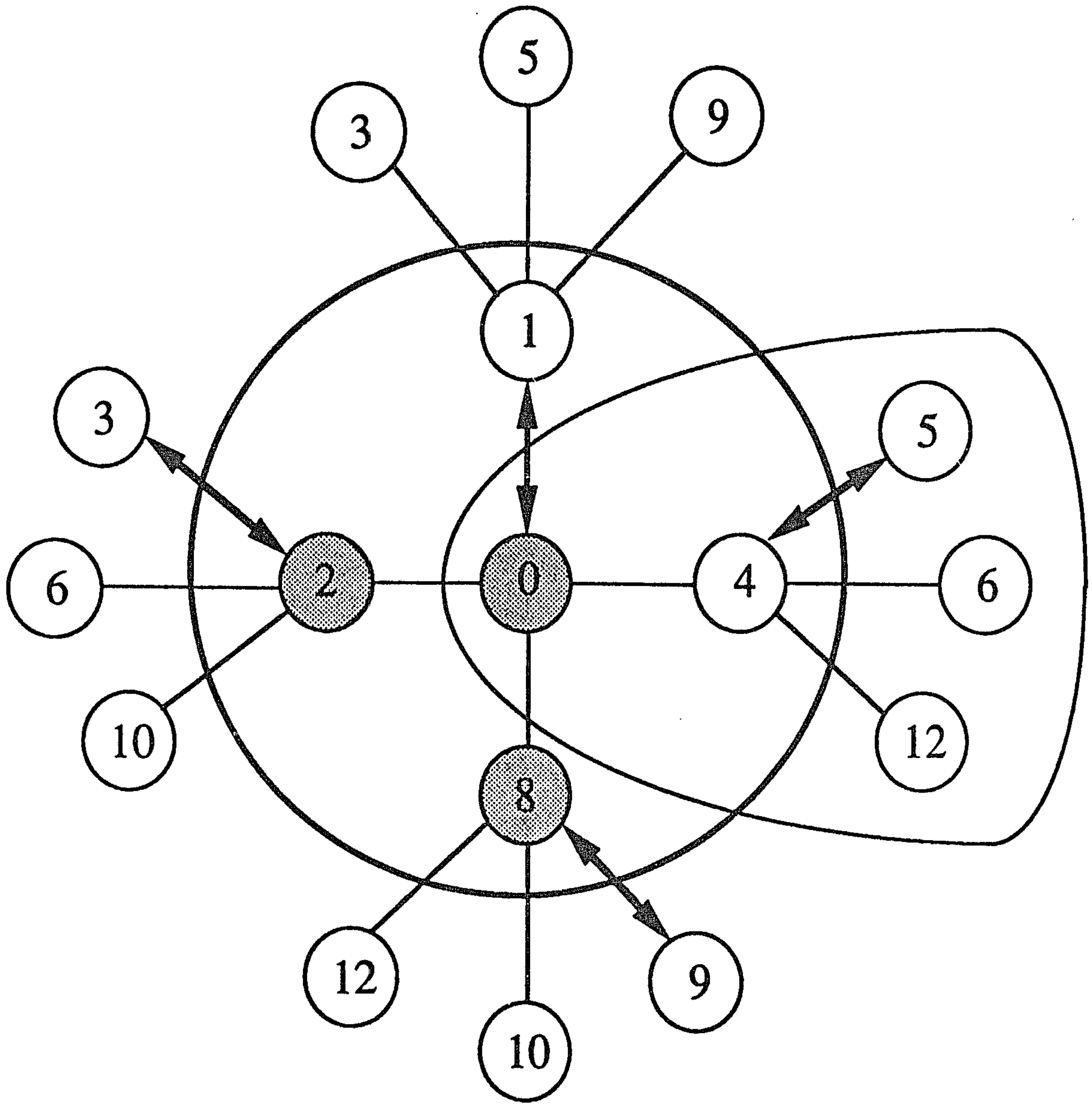


Figure 4.3: Solution for congestion problem

4.3.1 Construction of a Preferred List

Definition 4.1 *An n-dimensional hypercube, Q_n, is defined recursively as follows.*

1. *Q_0 is a trivial graph with one node, and*

2. *$Q_n = K_2 \times Q_{n-1}$, where K_2 is a complete graph with two nodes and $\times$ is the product operation on two graphs [73].*

The address of a Q_2 can be represented by a sequence of binary digits and $*$, $b_{n-1} \cdots b_2 * *$, where $*$ represents either 0 or 1. The four nodes, $b_{n-1} \cdots b_2 00$, $b_{n-1} \cdots b_2 01$, $b_{n-1} \cdots b_2 10$, $b_{n-1} \cdots b_2 11$, form a Q_2. Let the address of node i (denoted by N_i) be $i_{n-1} i_{n-2} \cdots i_0$, I_k be the unit vector in which all but k-th bit (which is set to '1') are '0', and $\oplus$ denote the bitwise exclusive-or operation. For example, every node, N_i, of an n-cube has n nodes within one hop and the address of each of these n nodes can be obtained by $N_i \oplus I_k$ for $k = 0, \ldots, n-1$.

Since the cost of transferring a task increases as the distance between the sender and receiver increases, it is naturally to order the nodes within one hop of the sender first, then the nodes within two hops, and so on. For convenience, the nodes within one hop of a node N_x are said to be in the *first entry* of N_x's preferred list, the nodes within two hops are in the *second entry*, and the nodes within m hops are in the *m^{th} entry.*

The nodes in all entries of N_i's preferred list are ordered as defined below.

Definition 4.2 *(1). The nodes in the first entry are ordered as $\{(i_{n-1} i_{n-2} \cdots i_0) \oplus I_j\}_{j=0}^{j=n-1}$.*

(2). The nodes in the second entry are ordered as $\{(i_{n-1} i_{n-2} \cdots i_0) \oplus I_j \oplus I_k\}$ ($j = 1, \ldots, n-2, 0$ and $j+1 \leq k \leq n-1$).

(3). The nodes in the third entry are ordered as $\{(i_{n-1} i_{n-2} \cdots i_0) \oplus I_j \oplus I_k \oplus I_\ell\}$ ($j = 1, \ldots, n-3, 0,$ $j+1 \leq k \leq n-2$ and $k+1 \leq \ell \leq n-1$).

(4). In general, the nodes in the k^{th} ($k \leq n$) entry of N_i's preferred list are ordered as $\{(i_{n-1} i_{n-2} \cdots i_0) \oplus I_{j_1} \oplus I_{j_2} \cdots \oplus I_{j_k}\}$ ($j_1 = 1, \ldots, n-k, 0,$ $j_1 + 1 \leq j_2 \leq n-k+1,$

$\cdots$, *and* $j_{k-1} + 1 \leq j_k \leq n - 1$).

The first i nodes in the first entry are often referred to as the most, second, ..., and i^{th} preferred nodes. Consider N_0 (labeled as 000) in a Q_3, the three nodes in N_0's first and second entries are ordered as $\{N_0 \oplus I_0,\ N_0 \oplus I_1,\ N_0 \oplus I_2\} = \{001, 010, 100\} = \{N_1, N_2, N_4\}$, and $\{N_0 \oplus I_1 \oplus I_2,\ N_0 \oplus I_0 \oplus I_1,\ N_0 \oplus I_0 \oplus I_2\} = \{110, 011, 101\} = \{N_6, N_3, N_5\}$. Similarly, the three nodes in N_3's (labeled as 011) first and second entries will be $\{010, 001, 111\} = \{N_2, N_1, N_7\}$, and $\{101, 000, 110\} = \{N_5, N_0, N_6\}$.

The ordering sequence in the first entry (shown in Definition 2) is one out of the $n!$ possible ways of ordering n nodes. I will show that all of such ordering ways are indistinguishable with respect to the capability and performance of load sharing: one can simply choose <u>any</u> one of them.

Ordering the nodes in the second entry is more involved than ordering the nodes in the first entry as discussed above. Consider N_0 in the previous example. Basically, one can copy the first entry nodes N_1, N_2, and N_4 to N_0's second entry. However, the most preferred nodes of the three nodes in N_0's first entry ($\{N_1, N_2, N_4\}$) will select the three nodes in N_1's first entry ($\{N_0, N_3, N_5\}$) as their most preferred nodes. Notice that N_0 will not transfer overflow tasks to nodes within two hops, i.e., in its second entry, unless N_1, N_2, N_4 are all either fully-loaded or overloaded. Since it is very likely that all these three nodes need to transfer tasks to their most preferred nodes ($\{N_0, N_3, N_5\}$), it is better for N_0 not to select N_3 first in its second entry to avoid collisions with N_0 and N_1 in transferring tasks to N_3. So, we consider the second preferred node of N_0, (i.e., N_2) first in generating N_0's second entry, and then the first entry of N_0's most preferred node (N_1).

Another issue in ordering the nodes in the second entry is to eliminate the duplicated nodes when copying the first entries of the nodes of N_0's first entry nodes. For example, the first entries N_1, N_2, and N_4 are $\{N_0, N_3, N_5\}$, $\{N_3, N_0, N_6\}$, and $\{N_5, N_6, N_0\}$, respectively. There are only three nodes, N_3, N_5, and N_6, need to be ordered in N_0's second entry. A systematic way to extract these distinct nodes is to set the index k of the second unit vector, I_k, in Definition 2 to be one larger than the first index j and increment k until it becomes $n-1$. So, the first index j of unit vector I_j can only be incremented up to $n-2$ and there is no need to consider the first entry of N_0's third preferred node (i.e., N_4) in this case.

Ordering the nodes in the third entry is similar to ordering the nodes in the second entry. According to Definition 2 nodes are ordered as $\{(i_{n-1}i_{n-2}\cdots i_0) \oplus I_j \oplus I_k \oplus I_\ell\}$. The first two indices j and k in I_j and I_k follow the same sequence as the ordering in the second entry except that the first (second) index j (k) can only be incremented up to $n-3$ ($n-2$), as explained above. The third index ℓ starts from $k+1$ and increments up to $n-1$ to avoid the duplication of nodes in N_0's the third entry. The fourth and higher entries are generated likewise. The preferred lists of all nodes in a 4-cube can be found in Fig. 2.2.

Once each node's preferred list is constructed, its buddy set can be formed by any required number of nodes counting from the top of its preferred list. (Note that the issue of determining the size of a buddy set has already been addressed in [25].) An overloaded node can then select an underloaded node from its buddy set based on the order of preference determined above. This selection can easily be implemented with a pointer to the first available node in the node's preferred list. If, albeit rare, an overloaded node cannot find any underloaded node from its buddy set, all of its tasks will be executed locally.

In what follows, we show that the above preferred lists can solve both the coordination and congestion problems.

Theorem 4.1 *Each node in an n-cube will be selected as the k^{th} preferred node by one and only one other node for $k = 1, \ldots, N (N = 2^n)$.*

The proof of this theorem is obvious according to Definition 2. Since the nodes in the first entry of each node are defined by $\{(i_{n-1}i_{n-2}\cdots i_0) \oplus I_j\}_{j=0}^{j=n-1}$, it is clear that the product of $(i_{n-1}i_{n-2}\cdots i_0) \oplus I_j$ on any two different nodes will be different for $j =$

$0, \ldots, n-1$. Moreover, ordering the nodes in the second and higher entries lists of each node is basically determined from the ordering of the first entry nodes. So, a node will be selected as the k^{th} preferred node by one and only one other node for $k = 1, \ldots, N$.

Since each node in a Q_n will be selected as the most preferred node by one and only one other node, the probability of an underloaded node being selected by more than one overloaded node is very small, thereby solving the coordination problem.

Before presenting Theorem 2, it is necessary to consider one special feature on hypercube structure, i.e., there does not exist a cycle which is composed of an odd number of nodes. For example, let the sequence of nodes $N_0, N_1, \ldots, N_k$ be part of a cycle. N_i and N_{i+1} are adjacent to each other for $i = 0, \ldots, k-1$. Using the $\oplus$ operation for $0 \leq i \leq k-1$, this sequence of nodes can be represented as follows. $N_{i+1} = N_i \oplus I_p$, and $N_{i+2} = N_{i+1} \oplus I_q = N_i \oplus I_p \oplus I_q$ for some $0 \leq p, q \leq n-1$. Then it is easy to show that N_{i+1} is one hop away from N_0, and N_{i+2} is either two hops away from N_0 if $p \neq q$ or equal to N_0 if $p = q$. Repeating this procedure, $N_{i+3} = N_0 \oplus I_p \oplus I_q \oplus I_r$ is either one hop away from N_0 if any two of p, q, r are equal, or three hops away from N_0 if p, q, r are all distinct integers. Generally, one can show that N_{i+j} will be an odd (even) number of hops away from N_0 if j is odd (even). In order to form a cycle, the start and end nodes must be one hop away from each other, so j can only be an odd number, thus making the total number of nodes in a cycle even.

Theorem 4.2 *The most preferred node of each node in a buddy set must come from a different buddy set if the buddy set size is no greater than* $\binom{n}{2}$.

Proof: Since the number of nodes in the first and second entries of an n-cube is $\binom{n}{1}$ and $\binom{n}{2}$, respectively, the buddy set of size no greater than $\binom{n}{2}$ will consist of only the nodes in the first or second entries. This theorem is proved in the following two steps.

1. No two nodes in the first or second entry can be adjacent to each other, otherwise one can form a cycle with an odd number of nodes, thus contradicting the feature of hypercube structure mentioned above. Since the most preferred node of any node is within one hop, any nodes in the first or second entry cannot select any other nodes in the same entry as its most preferred node.

2. Each node in the first entry will be adjacent to some other nodes in the second entry, but these adjacent nodes will not select each other as its most preferred node if the buddy set size is not greater than $\binom{n}{2}$ for the following reason. As stated earlier, the most preferred node of each node in the first entry will be the nodes in the first entry of the most preferred node. According to Definition 2, the first entry of the most preferred node is the last entry to be considered in generating of the second entry for each node. Since there are $\binom{n}{1}$ nodes in the first entry, if the buddy set size is not greater than $\binom{n}{2}$, then only the first $\binom{n}{2} - \binom{n}{1}$ nodes in the second entry will be included in the buddy set. Thus none of the nodes in the first entry will select any nodes in the second entry which are included in the buddy set as their most preferred node. □

If system load is well balanced, the probability of staying in overloaded state is much smaller than in underloaded state. So, an overloaded node is most likely to transfer a overflow task to its most preferred node; the overloaded nodes in a buddy set will spread their overflow tasks out to many different buddy sets instead of overloading the nodes in its own buddy set, thus solving the congestion problem. As concluded in [25], buddy sets with 10 to 15 nodes perform well in most cases, so the buddy set size of $\binom{n}{2}$ in Theorem 4.2 will not alter the usefulness of the preferred lists in practice for hypercube multicomputers larger than Q_5.

As mentioned earlier, ordering the nodes in the first entry (Definition 2) is one out of $n!$ possible ways. It is easy to show that all these ordering ways are indistinguishable with respect to the properties of Theorem 4.1 and 4.2.

From Definition 2, the nodes in the first entry are ordered as $\{(i_{n-1}i_{n-2}\cdots i_0) \oplus I_j\}_{j=0}^{j=n-1}$. One can generate another set of preferred lists by interchanging the incrementing sequence of index j of I_j. However, such preferred lists will also satisfy the properties of Theorem 4.1 and 4.2. So, they are indistinguishable with respect to the properties of Theorem 4.1 and 4.2.

4.4 Calculation of the Number of Task Collisions

The location policy is one of the key issues that determine the performance of LS schemes. A good location policy should be able to move the overflow tasks from overloaded nodes to underloaded nodes in such a way that each underloaded node shall not exceed its capability due to transferred tasks; otherwise, some of the underloaded nodes may become overloaded and need to transfer out some of the received tasks, thus possibly increasing the network traffic and the task transfer delay without improving the performance.

So, we develop a model using the number of task collisions to analyze the goodness of a location policy. Specifically, if $n+1$ overflow tasks are sent simultaneously to an underloaded node, n task collisions will result. This quantification of task collision is to reflect the severity of the coordination problem. We want to devise a location policy so that each overloaded node may find an underloaded receiver without any conflicts with other overloaded nodes. Although acceptance of more than one overflow task at a time may not necessarily make an underloaded node overloaded, the average response time in such a case will be larger than the case when each overloaded node can find an underloaded node which is not being chosen by any other overloaded node. For example, if an underloaded node can complete only one overflow task in time, then the rest of simultaneously transferred–in tasks will miss their deadlines or must be retransferred. Under this model, the best location policy is the one that results in no task collision at all.

The number of task collisions is strongly related to the performance of any location policy, if an optimal location policy is defined based on the fact that each underloaded node must not receive tasks more than its capacity at any moment. In such cases the optimal location policy is the one that results in no task collision; otherwise, some of underloaded nodes may exceed its capacity and have to retransfer received tasks.

Let the *capacity* of an underloaded node be defined as the maximum number of tasks that a node can receive without becoming overloaded. Assuming that the number of underloaded nodes is always greater than or equal to the number of overloaded nodes, the total number of overflow tasks is less than the combined capacities of all underloaded nodes. One can show that the only condition to guarantee that each underloaded node not to exceed its capacity by accepting transferred tasks is to avoid task collisions. Since each underloaded node can accept at least one task, it will not exceed its capacity when there is no task collision (each underloaded node will receive at most one task at a time). As soon as an underloaded node receives a transferred task, it will update and multicast its load state to the nodes in its buddy set. If there are task collisions, some of the underloaded nodes may receive more transferred tasks than their capacity, thus needing to retransfer some of the received tasks. So to guarantee each underloaded node not to receive more transferred tasks than its capacity is to avoid task collisions.

In the rest of this section, we shall derive the number of task collisions for the proposed location policy as a function of the total number of nodes and the number of overloaded nodes. To simplify the analysis, we assume that only U– or V–state nodes exist in the system. We will show later that this analysis can be easily adapted to the case with F–state nodes. The conditions that result in no task collision are stated in the following two lemmas.

Lemma 4.1 *If at most two nodes in a Q_2 are overloaded, there will be no task collisions in the Q_2.*

Proof: If there is only one overloaded node in a Q_2, this node will transfer an overflow task to its most preferred node which is underloaded. If there are two overloaded nodes in the Q_2, these two nodes will both select their first or second preferred nodes to be the receivers. In either case, no task collision will occur according to Theorem 4.1. □

Note that without using the preferred list, there is a 50% chance that two overloaded nodes will simultaneously transfer tasks to the same underloaded node in a Q_2, thus resulting in one task collision. Let $N = 2^n$ and k be the number of overloaded nodes in an n-cube. From Lemma 4.1, the number of ways these k overloaded nodes can be distributed in the system without resulting in any task collision is given by:

$$\sum_{i=0}^{\lceil k/2 \rceil} \frac{(k-i)!}{i!\,(k-2i)!} \binom{N/4}{k-i} \binom{4}{1}^{k-2i} \binom{4}{2}^{i} = \sum_{i=0}^{\lceil k/2 \rceil} \binom{N/4}{N/4-k+i} \binom{k-i}{i} \binom{4}{1}^{k-2i} \binom{4}{2}^{i}$$
$$= \sum_{i=0}^{\lceil k/2 \rceil} \binom{N/4}{N/4-k+i,\ i,\ k-2i} \binom{4}{1}^{k-2i} \binom{4}{2}^{i} \quad (4.1)$$

where $\binom{a}{b} = \frac{a!}{(a-b)!\,b!}$ and $\binom{a}{b,\ c,\ d} = \frac{a!}{b!\,c!\,d!}$.

For convenience, let $\Lambda(N,k) = \sum_{i=0}^{\lceil k/2 \rceil} \binom{N/4}{N/4-k+i,\ i,\ k-2i} \binom{4}{1}^{k-2i} \binom{4}{2}^{i}$.

Eq. (4.1) can be explained as follows. Let x and y be the number of Q_2's that have one or two overloaded nodes in a Q_n, respectively. Then $x + y \leq 2^{n-2}$ ($= N/4$) and $2x + y = k$ when $k \leq 2^{n-1}$, for $x = 0 \ldots \lceil k/2 \rceil$, $y = k - 2x$. The number of ways of distributing k overloaded nodes in the system without any task collision is equivalent to the number of ways of choosing $x + y$ Q_2's out of the total number of Q_2's ($N/4$) in a Q_n; thus giving the $\binom{N/4}{k-i}$ term in Eq. (4.1). Since there are many possible combinations of x and y, we need to sum up all these possible combinations. Furthermore, the number of ways of assigning the overloaded nodes to the selected Q_2's is exactly the same as the binomial formula, thus giving the $\binom{4}{1}^{k-2i} \binom{4}{2}^{i}$ term in Eq. (4.1).

Lemma 4.2 *If all four nodes in a Q_2 are overloaded but none of the nodes in its most preferred Q_2 is overloaded, no task collision will occur to this Q_2.*

Proof: Even if all four nodes are overloaded in a Q_2, each of them will select one node from its most preferred Q_2. Since none of the nodes in the latter Q_2 is overloaded, no task collision will occur. □

From Lemma 4.2, the number of ways k overloaded nodes can be distributed over N nodes without any task collision is

$$\sum_{a=1}^{\lfloor k/4 \rfloor} \binom{N/4}{a} \times \left[\sum_{i=0}^{k'/2} \binom{N'/4}{k'-i} \binom{k'-i}{i} \binom{4}{1}^{k'-2i} \binom{4}{2}^{i} \right]$$

$$= \sum_{a=1}^{\lfloor k/4 \rfloor} \binom{N/4}{a} \times \left[\sum_{i=0}^{k'/2} \binom{N'/4}{N/4-k'-i,\, i,\, k'-2i} \binom{4}{1}^{k'-2i} \binom{4}{2}^{i} \right] \tag{4.2}$$

where a is the number of groups with four overloaded nodes, $N' = N - 8a$, and $k' = k - 4a$. The first term $\binom{N/4}{a}$ in Eq. (4.2) is the number of ways of selecting a Q_2's out of the total $N/4$ Q_2's and all four nodes in these selected Q_2s are overloaded. The rest of overloaded nodes, $k' = k - 4a$, can be distributed among the rest of nodes, $N' = N - 8a$, as shown in Eq. (4.1).

Combining Eqs. (4.1) and (4.2), the probability, P_0, of no task collision given N and k, can then be calculated by:

$$P_0 = \frac{\Lambda(N,k) + \sum_{a=1}^{\lfloor k/4 \rfloor} \binom{N/4}{a} \Lambda(N',k')}{\binom{N}{k}}, \quad \text{where } N' = N - 8a, k' = k - 4a. \tag{4.3}$$

Theorem 4.3 *At least three nodes must be overloaded to result in task collisions, and these three nodes must include a node, and both its first and second preferred nodes; otherwise, no task collision will occur.*

Proof: If the number of overloaded nodes is less than three, or if there are three overloaded nodes distributed over at least two Q_2's, then no collision will occur according to Lemma 4.1. If all three overloaded nodes reside in the same Q_2, one can show that these three nodes must include a node and both its first and second preferred nodes, and in such a case, a task collision will always occur.

Since there is no replication of nodes in a preferred list, a node and its first and second preferred nodes must all be different. Since there are only four different ways to choose three out of the four nodes in a Q_2, each of these choices must include a node and its first and second preferred nodes.

The last part of this proof is to show that if a node and its first two preferred nodes are overloaded, a task collision will always occur. According to Definition 2, the first and second preferred nodes of a node N_i are $i_{n-1}i_{n-2}\cdots i_0 \oplus I_0$ and $i_{n-1}i_{n-2}\cdots i_0 \oplus I_1$. Since all these nodes are overloaded, each of them will try to transfer a task to an underloaded node in its preferred list. It is easy to show that N_i will transfer an overflow task to node $i_{n-1}i_{n-2}\cdots i_0 \oplus I_2$, while node $i_{n-1}i_{n-2}\cdots i_0 \oplus I_0$ (N_i's first preferred node) will transfer an overflow task to node $i_{n-1}i_{n-2}\cdots i_0 \oplus I_0 \oplus I_1$, and node $i_{n-1}i_{n-2}\cdots i_0 \oplus I_1$ (N_i's second preferred node) will transfer an overflow task to node $i_{n-1}i_{n-2}\cdots i_0 \oplus I_1 \oplus I_0$. Since $I_0 \oplus I_1 = I_1 \oplus I_0$, the first and second preferred nodes of N_i will simultaneously transfer tasks to the same node $i_{n-1}i_{n-2}\cdots i_0 \oplus I_0 \oplus I_1$, thus resulting in a task collision. $\square$

Corollary 4.1 *If the number of overloaded nodes is greater than 3 and includes the pattern described in Theorem 4.3, task collisions will always occur; otherwise, no collision will occur.*

Task collision will occur with the following patterns of overloaded nodes, or combinations thereof:

Pattern 1: three overloaded nodes in a Q_2

$\Longrightarrow$ one task collision.

Pattern 2: four overloaded nodes in a Q_2 whose most preferred Q_2 has one overloaded node

$\Longrightarrow$ one task collision.

Pattern 3: four overloaded nodes in a Q_2 whose most preferred Q_2 has two overloaded nodes

$\Longrightarrow$ two task collisions.

Given N and k, the probability of resulting in a task collision is given by

$$P_1 = \frac{\binom{N/4}{1}\binom{4}{3}\Lambda(N',k') + \binom{N/4}{2}\binom{4}{1}\Lambda(N'',k'')}{\binom{N}{k}} \tag{4.4}$$

where $N' = N-4, N'' = N-8, k' = k-3$, and $k'' = k-5$. The first (second) term in Eq. (4.4) is the number of ways to select one Q_2 out of the total $N/4$ Q_2's and assign three (four) overloaded nodes in this Q_2 times the number of ways of distributing the rest of overloaded nodes without any task collision.

When two or more task collisions occur, all combinations of these patterns need to be considered. For example, ℓ collisions can result from any combination of a groups of pattern 1, b groups of pattern 2, and c groups of pattern 3, such that $a+b+2c=\ell$ and $a,b,c \in \{0,1,\ldots,\ell\}$. Given N and the number, ℓ, of collisions, Algorithm 4.1 determines the total number of combinations of these patterns.

Algorithm 4.1

```
For $t_{rej} := 1$ to $\ell$ do
    for $i := 0$ to $\lfloor \frac{t_{rej}}{2} \rfloor$ do
        for $j := t_{rej} - 2i$ to 0 do
            $Pattern_1 \longleftarrow j$;
            $Pattern_2 \longleftarrow t_{rej} - 2i - j$;
            $Pattern_3 \longleftarrow i$;
            $N_{over} \longleftarrow Pattern_1 \times 3 + Pattern_2 \times 5 + Pattern_3 \times 6$;
            $j \longleftarrow j - 1$;
        end_do
        $i \longleftarrow i + 1$
    end_do
end_do
```

Although many combinations of the above three patterns can result in the same number of task collisions, they may require a different number of overloaded nodes which was given as N_{over} in Algorithm 4.1. Once combinations of these patterns are determined, the probability of resulting in ℓ task collisions, denoted by P_ℓ, can be calculated by:

$$P_\ell = \frac{\sum\limits_{\text{all } a\,b\,c} \binom{N/4}{a}\binom{N/4-a}{b}\binom{N/4-a-b}{c}\binom{a+b+c}{a\;b\;c}\Lambda(N',k')}{\binom{N}{k}} \tag{4.5}$$

where $\ell = 1, \ldots, t_{rej}$, $N' = N - 4\ell$, and $k' = k - N_{over}$.

It should be noted that in the above analysis only U– and V–state nodes are considered. Eqs. (4.4) to (4.5) can be easily modified to derive task collisions in the presence of F–state nodes. Let F and k be the number of nodes in F– and V–state, respectively. One can add F to k as the total number of V–state nodes and calculate the task collisions by Eq. (4.5). The actual number of task collisions will be $[k/(F+k)]^2$ times Eq. (4.5), because a task collision will occur only when two V–state nodes simultaneously send tasks to the same U–state node.

4.5 Comparative Performance Evaluation

The performance of the proposed location policy is compared to that of other policies based on probing, random selection, bidding and drafting algorithms. The random selection policy is exactly the same as the proposed location policy except that it does not use preferred lists. Since each node collects, via state–change multicasts, the state information of other nodes in its buddy set, V–state node can randomly select one of the U–state nodes and transfer an overflow task to that node. Note that this is a typical location policy when a node's LS is restricted to its neighboring nodes. Moreover, the performance obtained from this method can be used for comparison with the approach of using the preferred lists. In the probing policy, on the other hand, a V–state node will randomly probe the nodes in a buddy set to find a receiver. The V–state node will transfer an overflow task to the node it probed if it happens to be underloaded; otherwise, it will repeat the probing process with another node, up to a total of 5 probes. The reason for using up to 5 probes is based on the finding in [40]. The buddy set size is chosen to be 10, since the performance improvement by increasing the buddy set size beyond 10 was shown to be insignificant [25].

In the bidding algorithm [33, 34], the V–state nodes will broadcast the bids to all other nodes and the U–state nodes will send their bids to the V–state nodes. If every U–state node has the same load state, the V–state node will randomly accept one of the bids received; this is similar to the random selection policy. However, if the U–state nodes have different load states, the node with the least workload will be selected by all V–state nodes, so some of the transferred tasks need to be retransferred if the total number of overflow tasks exceeds the capacity of this U–state node. In the drafting algorithm [35], two more messages will be sent between the U–state and V–state nodes to avoid 'dumping' many overflow tasks to the same U–state node. So, the U–state node can reject selection by the rest of V–state nodes when its state moves to F, thus avoiding task retransfers.

The coordination problem is analyzed when the V–state nodes are randomly distributed in the system. It is necessary to introduce the following notation:

- $N = 2^n$: total number of nodes in the n–cube system
- F : total number of fully-loaded nodes in the system
- k_s : total number of overloaded nodes in the system
- k_b : average number of overloaded nodes in a buddy set
- σ : size of a buddy set
- c_t $(\overline{c_t})$: (average) number of task collisions
- $P_i(k)$: probability of i task collisions when there are k overloaded nodes
- $P_0^{(i)}$: probability for the i^{th} overloaded node to find an underloaded node within 5 probes.

4.5.1 Probing policy

Under this policy, an overloaded node randomly probes up to five nodes and transfers an overflow task to the first underloaded node found during the probing process. Based on the location of the nodes to be probed, two cases are considered: the entire system and a buddy set.

Probing the Entire System

In order for each V–state node to locate U–state node whenever U–state node accepts an overflow task, it will refuse, even if it is still underloaded, to accept any more overflow tasks unless it is instructed otherwise. If a V–state node cannot find a U–state node which has not yet received any overflow task within five probes, it can either process this task locally or transfer this task to one of the U–state nodes which had already accepted one overflow task. In either case, a task collision is considered to have occurred in our model. The probability that each of k_s overloaded nodes can find an underloaded node is given by:

$$\begin{aligned} P_0^{(1)} &= \sum_{i=1}^{5} \left[\prod_{j=1}^{i-1} \frac{F+k_s-j}{N-j} \right] \frac{N-F-k_s}{N-i} \\ P_0^{(m)} &= \sum_{i=1}^{5} \left[\prod_{j=1}^{i-1} \frac{F+k_s-j+m-1}{N-j} \right] \frac{N-F-k_s-m+1}{N-i} \qquad \text{for } m=1,\ldots,k_s. \end{aligned}$$

Then the probability of $r_t = m$ $(1 \leq m \leq k_s)$ can be calculated as:

$$\begin{aligned} P_0(k_s) &= \prod_{i=1}^{k_s} P_0^{(i)} \\ P_1(k_s) &= \sum_{i=1}^{k_s} \left[\prod_{j=1,j\neq i}^{k_s} (1-P_0^{(i)})\, P_0^{(j)} \right] \\ P_m(k_s) &= \sum_{x_1=1}^{k_s} \cdots \sum_{x_m=x_{m-1}}^{k_s} \left[\prod_{y=1,y\neq x_1,\cdots,y\neq x_m}^{k_s} (1-P_0^{(x_1)}) \cdots (1-P_0^{(x_m)})\, P_0^{(y)} \right]. \end{aligned} \quad (4.6)$$

Although $\overline{c_t} = \sum_{i=1}^{k_s} i\,P_i(k_s)$ can be calculated by using Eq. (4.6), the nested summations in these equations require a prohibitive amount of computation. The required computation can be reduced significantly if there is one representative, approximate value (e.g., average value) for all $P_0^{(i)}$'s. Eq. (4.6) can then be simplified as

$$P_i(k_s) = \binom{k_s}{i} \; (1 - P_0^{(j)})^i \; \left(P_0^{(j)}\right)^{k_s - i} \quad \text{for } i = 1, \ldots, k_s. \tag{4.7}$$

To get a better approximation, j can be set to $k_s/2$, or $P_0^{(j)}$ can be replaced by the average value of all these terms. Comparison between the approximate and simulation (exact) results for 6- and 8- cubes is summarized in Table 4.1. The approximation by using the average value is shown to be close to the simulation results except when the number of overloaded nodes reaches $N/2$. In such a case the median value approximation works better.

Although Eq. (4.6) is derived in a sequential manner, it gives the same result as when all overloaded nodes perform the probing procedure concurrently. Suppose an underloaded node receives more than one probe at the same time, then this node should reply to only one probe when it is in U–state whereas it must reply to all other probe requests when it is in F– or U– state; otherwise, this node may receive more than one overflow task and may become overloaded. So, although different nodes run the probing procedure concurrently in practice, the probability of probe collisions remains the same as the case when the probing is done sequentially by the nodes.

Random Probing in a Buddy Set

When the nodes to be probed by an overloaded node are restricted to its own buddy set, the overloaded node may not be able to find an underloaded node if there are more than 5 overloaded nodes in the buddy set. So, the first step in calculating $\overline{c_t}$ is to derive the number of overloaded nodes each with more than 5 overloaded nodes in their buddy set.

Let $\chi(m, k_s)$ be the number of nodes each with m overloaded nodes in their buddy set, where k_s is the total number of overloaded nodes in the system, and let $\eta = (N - \sigma)/N$. $\chi(m, k_s)$ is a recursive function given by:

$$\chi(m, k_s) \;\; = \;\; \chi(m, k_s - 1) - (1 - \eta)\, \chi(m, k_s - 1) + (1 - \eta)\, \chi(m - 1, k_s - 1)$$

$$= \eta\, \chi(m, k_s - 1) + (1-\eta)\, \chi(m-1, k_s - 1) \tag{4.8}$$

$$\text{for } m = 0, \ldots, \sigma - 1$$

$$\chi(\sigma, k_s) = \chi(\sigma, k_s - 1) + (1-\eta)\, \chi(\sigma - 1, k_s - 1) \tag{4.9}$$

Initial conditions : $\chi(0,0) = N$

$\chi(m, k_s) = 0, \;\; m = 0, \ldots, \sigma, \;\; k_s = 0, 1, \ldots, m-1$

$\chi(m, k_s) = 0, \;\; \text{if } m < 0.$

$\chi(m, k_s)$ in Eq. (4.8) is obtained by introducing one more V–state node in the system with $k_s - 1$ overloaded nodes. Since these overloaded nodes may be formed with either $m-1$ or m V–state nodes in the buddy set with probability $1-\eta$, the former case will add up to $\chi(m, k_s)$ and the latter case will move to $\chi(m+1, k_s)$, thus giving the third and second terms in Eq. (4.8). The rest of nodes, $\chi(m, k_s - 1)$, will become $\chi(m, k_s)$ as the first term in Eq. (4.8).

Eqs. (4.8) and (4.9) can be solved by successively decrementing k_s until the initial conditions are reached, leading to the following closed-form equations:

$$\chi(0, k_s) = N \left(\frac{N-\sigma}{N}\right)^{k}_{s} \tag{4.10}$$

$$\chi(1, k_s) = \sigma\, k_s \left(\frac{N-\sigma}{N}\right)^{k_s - 1} \tag{4.11}$$

$$\chi(2, k_s) = \binom{k_s}{2}\, \sigma\, (1-\eta) \left(\frac{N-\sigma}{N}\right)^{k_s - 2} \tag{4.12}$$

$$\chi(\sigma, k_s) = \frac{\sigma}{(\sigma-1)!}\, (1-\eta)^{\sigma-1} \left[\sum_{j=0}^{k_s-1} \eta^j\, j^{\sigma-1}\right] \tag{4.13}$$

$$\chi(m, k_s) \approx \frac{10}{m!}\, (1-\eta)^{m-1}\, \eta^{k_s - m}\, k_s^m \;\; \text{for } m = 3, 4, \ldots, \sigma - 1. \tag{4.14}$$

Note that Eqs. (4.10) to (4.13) give the exact solutions while Eq. (4.14) gives an approximate solution in which $\sum_{j=2}^{k_s-1} j^a$ is approximated by $\frac{j^{a+1}}{a+1}$. Thus

$$\overline{c_t} = \frac{k_s}{N} \left[\sum_{m=5}^{\sigma} \chi(m, k_s) \sum_{i=0}^{4} \prod_{j=1}^{i} \frac{F + k_s - j}{N - j}\, \frac{N - F - k_s}{N - i}\right]. \tag{4.15}$$

4.5.2 Random Selection

Under this policy, an overloaded node randomly selects one underloaded node to which an overflow task is transferred. There are $m-1$ task collisions if $m>1$ overloaded nodes simultaneously transfer tasks to the same underloaded node. Again, two cases need to be considered on the basis of the location of nodes to be selected: the entire system or a buddy set.

Random Selection in the Entire System

For convenience, define a function which gives the probability of no task collision:

$$P_0(x,y,s) = \frac{(x-y-s)\,(x-y-s-1)\cdots(x-2y-s+1)}{(x-y)^y} = \frac{y!\binom{x-y-s}{y}}{(x-y)^y},$$

where x and y can be the total number of nodes and V–state nodes in the system, respectively, and s is generally related to the number of possible patterns that will result in the same number of task collisions. When $s=0$, $P_0(N,k,0)$ becomes

$$P_0(k_s) = \frac{(N-k_s)\,(N-k_s-1)\cdots(N-2k_s+1)}{(N-k_s)^{k_s}} = \frac{k_s!\binom{N-k_s}{k_s}}{(N-k_s)^{k_s}}.$$

This is the probability of no task collision in a system with N nodes and k_s overloaded nodes, where the numerator represents the patterns of task transfer without any task collision, while the denominator represents all possible patterns of task transfer. Since there will be one collision if any two out of k overloaded nodes simultaneously transfer tasks to the same underloaded node, the probability of resulting in one collision is shown to be:

$$\begin{aligned} P_1(k_s) &= \frac{\binom{k_s}{2}}{N-k_s}\,\frac{(N-k_s-1)\cdots(N-2k_s+2)}{(N-k_s)^{k_s-2}} \\ &= \frac{\binom{k_s}{2}}{N-k_s}\,\frac{(k_s-2)!\binom{N-k_s-1}{k_s-2}}{(N-k_s)^{k_s-2}} = \frac{\binom{k_s}{2}}{N-k_s}\,P_0(N-2,k_s-2,1). \end{aligned}$$

Similarly, when x overloaded nodes simultaneously transfer tasks to the same underloaded node, $x-1$ collisions will occur with a probability:

$$\begin{aligned} P_{x-1}(k_s) &= \frac{\binom{k_s}{x}}{(N-k_s)^{x-1}} \frac{(N-k_s-1)\cdots(N-2k_s+2)}{(N-k_s)^{k_s-x}} \\ &= \frac{\binom{k_s}{x}}{(N-k_s)^{x-1}} \frac{(k_s-x)!\binom{N-k_s-1}{k_s-x}}{(N-k_s)^{k_s-x}} \\ &= \frac{\binom{k_s}{x}}{(N-k_s)^{x-1}} P_0(N-x, k_s-x, 1) \quad \text{where } x = 2,3,\ldots,k_s. \end{aligned} \quad (4.16)$$

However, in the cases of two or more collisions, many possible task transfer patterns need to be considered. For example, two collisions will occur when either one group of three overloaded nodes, or two groups of two overloaded nodes each transfer tasks to one underloaded node. A sequence, $\{x_i\}_{i=1}^{n}$, of positive integers are used to represent a task transfer pattern which results in m collisions. A transfer pattern which results in c_p collisions can be represented as

$$pattern_a \longrightarrow \underbrace{m}_{x_{m-1}}\underbrace{m-1}_{x_{m-2}}\cdots\underbrace{3\cdots3}_{x_2}\underbrace{2\cdots2}_{x_1} \quad \text{where } m = c_p+1, x_{m-1}=1.$$

We then have $c_p = \sum_{i=1}^{m-1} ix_i$ and $k_p = \sum_{i=1}^{m-1}(i+1)x_i$. Although many transfer patterns may result in the same number of collisions, the number of overloaded nodes, k_p ($\le k_s$), may not be the same in these patterns. Eq. (4.16) can be generalized to calculate the probability of having c_p collisions under *pattern*$_a$:

$$\begin{aligned} P_{c_p}(k_s) &= \frac{\binom{k_s}{2}\cdots\binom{k_s-2(x_1-1)}{2}\cdots\binom{k_s-\sum_{i=1}^{m-1}(i+1)\,x_i}{m}\cdots\binom{k_s-k_p+m}{m}}{(N-k_s)\,\cdots\,(N-k_s-m+1)} \\ &\quad \times P_0(N-k_p, k_s-k_p, \sum_{i=1}^{m} x_i). \end{aligned} \quad (4.17)$$

Generating All Possible Patterns

When using Eq. (4.17) to calculate the average number of task collisions under the random selection policy, one must generate all possible patterns that result in the same number of collisions. We will show that the number of patterns to be considered grows rapidly as the number of collisions increases. Let function $F(c_p)$ be the total number of patterns that result in c_p collisions under the random selection policy. Also, let function $f_m(c_p)$ be the number of patterns that result in c_p collisions with m as the first integer — called the *leading number* — in the patterns.

Lemma 4.3 $F(c_p) = 1 + \sum_{m=3}^{c_p+1} f_m(c_p)$.

Proof: When $x_i = 0$ for $i \geq 2$, there will be only one pattern with $x_1 (= c_p)$ 2's. When $x_1, x_2 \neq 0$ and $x_i = 0 \; \forall \; i \geq 3$, the number of patterns is $\lfloor \frac{c_p}{2} \rfloor$ and determines $f_3(c_p)$. Similarly, when $x_j \neq 0 \; \forall \; j < k$ and $x_i = 0 \; \forall \; i > k$, the number of patterns determines $f_k(c_p)$. So, the total number of patterns for a given c_p is $1 + \sum_{m=3}^{c_p+1} f_m(c_p)$. □

Lemma 4.4 $f_m(c_p) = 1 + \sum_{x=3}^{m} f_x(c_p - m + 1)$ *for* $m = 3, \ldots, c_p + 1$.

Proof: Since the largest leading number in the patterns specified by $f_m(c_p)$ is m, the first pattern will be $\underbrace{m}_{1} \underbrace{2 \cdots 2}_{c_p - m + 1}$. Other than the first integer m, the rest of $(c_p - m + 1)$ 2's can have leading numbers from 3 to m, so the total number of patterns in $f_m(c_p)$ is equal to $1 + \sum_{x=3}^{m} f_x(c_p - m + 1)$. □

According to Lemma 4.4, $f_m(c_p)$ can be obtained by successively replacing $f_m(c_p - m + 1)$ by the functions with the leading number $< m$:

$$\begin{aligned} f_{x+1}(m) &= \lfloor \frac{k_s}{x} \rfloor + \sum_{j=0}^{x-2} \sum_{i=1}^{\lfloor \frac{k_s}{x} \rfloor} f_{x-j}(k_s - xi) \; = \; \lfloor \frac{k_s}{x} \rfloor + \sum_{i=1}^{\lfloor \frac{k_s}{x} \rfloor} f_{x-1}(k_s - xi) \\ &+ \sum_{i=1}^{\lfloor \frac{k_s}{x} \rfloor} f_{x-2}(k_s - xi) + \cdots + \sum_{i=1}^{\lfloor \frac{k_s}{x} \rfloor} f_3(k_s - xi). \end{aligned} \tag{4.18}$$

A few examples of approximate, closed-form expressions for these functions are:

$$\begin{aligned}
f_3(k_s) &= \lfloor \frac{k_s}{2} \rfloor \\
f_4(k_s) &\approx \frac{1}{12}\, k_s(k_s+1) \\
f_5(k_s) &\approx \frac{1}{4!}\left(\frac{1}{3!}\, k_s^3 + \frac{5}{2!}\, k_s^2 + 3!\, k_s\right) \\
f_6(k_s) &\approx \frac{1}{5!}\left(\frac{1}{4!}\, k_s^4 + \frac{9}{3!}\, k_s^3 + \frac{22}{2!}\, k_s^2 + 4!\, k_s\right) \\
f_x(k_s) &\approx \frac{1}{(x-1)!}\left(\frac{1}{(x-2)!}\, k_s^{x-2} + \frac{x\,(x-3)}{2\,(x-3)!}\, k_s^{x-3} + \cdots + (x-2)!\, k_s\right).
\end{aligned}$$

However, it is still tedious to derive $f_m(k_s)$ for the patterns with leading numbers greater than 10. The following theorem shows that $F(k_s)$ is a polynomial of k_s.

Theorem 4.4 *$F(k_s)$ is at least $O(k_s^x)$, where x can be any positive integer greater than 2.*

Proof: Consider the first term in functions of $f_m(k_s)$ in Lemma 4.3 and the approximate closed-form expressions derived above. $F(k_s) \approx 1 + \sum_{x=3}^{k_s+1} \frac{k_s^{x-2}}{(x-1)!\,(x-2)!}$. Since the summation runs from $x = 3$ to $x = k_s+1$, one can choose one term from this summation where $x \ll k_s$, such that the summation will become at least $O(k_s^x)$. When $k_s \to \infty$, x will also get large, and thus, the theorem follows. □

Approximate Formula

Although the above method can give the exact number of task collisions in the system, it is, in practice, too tedious to calculate the related equations, because the number of possible patterns grows much faster than the number of collisions. Moreover, these patterns must be stored in memory; even for a moderate range of c_p (a few hundred) the required storage space exceeds by far the capacity of any existing computer systems (several hundred giga bytes). Thus, a good approximation to the number of collisions is called for.

Since task collisions occur when more than two overloaded nodes simultaneously transfer tasks to the same underloaded node, one can choose i overloaded nodes out of the total k_s overloaded nodes for $i = 2, \ldots, k_s$ such that these i overloaded nodes simultaneously choose the same underloaded node for task transfer. The rest of overloaded nodes $(k_s - i)$ can choose any other underloaded nodes except those already chosen. The average number of collisions can then be calculated by summing all possible values of i as:

$$\overline{c_t} = \sum_{i=2}^{k_s} \binom{k_s}{i} \frac{i-1}{(N-k_s)^{i-1}} \left(\frac{N-k_s-1}{N-k_s}\right)^{k_s-i}. \tag{4.19}$$

Eq. (4.19) gives an approximate figure in the sense that the rest of overloaded nodes (other than the chosen nodes) may also simultaneously transfer tasks to some other underloaded nodes and result in collisions, but are not included in the above equation. When compared with the simulation (exact) results, Eq. (4.19) is shown to be a good approximation (see Table 4.2).

Random Selection in a Buddy Set

Under this policy, an overloaded node randomly selects one underloaded node within its buddy set. A collision will occur only when the buddy sets of two overloaded nodes overlap. So, the first step is to determine the intersection (overlap) of two buddy sets. According to [25], buddy sets of 10 to 15 nodes yield good results, and thus, the overlap between the buddy sets of any two nodes can be approximated under the assumption that each node's buddy set is formed with only its immediate neighbors.

Theorem 4.5 *The number of ways that the buddy sets of any two nodes overlap is* $\binom{n}{2}$ *and only two nodes overlap between the two buddy sets.*

Proof: If a buddy set is composed of only immediate neighbors of the corresponding node, then the nodes in a node i_0's buddy set can be determined by $i_0 \oplus I_p$ $(0 \le p \le n-1)$ according to Definition 2. The buddy sets of nodes $i_0 \oplus I_j \oplus I_k$ $(0 \le j \le n-1$ and $j+1 \le k \le n-1)$ will overlap node i_0's buddy set when $p = j$ or $p = k$, because $(i_0 \oplus I_j \oplus I_k) \oplus I_k = i_0 \oplus I_j$. So, a total of $\binom{n}{2}$ nodes will overlap node i_0's buddy set, and only two nodes overlap, i.e., when $p = j$ and $p = k$. □

Theorem 4.6 *The number of ways the buddy sets of k nodes overlap is $2(n-k+1)$ for $k = 3, \ldots, n$, and only one node overlap between the k buddy sets.*

Proof: From Theorem 4.5, if the buddy sets of k nodes overlap, these nodes must contain a node i_0 and the nodes which are two hops away from node i_0. One can choose the first $k-1$ nodes as i_0, $i_0 \oplus I_p \oplus I_a$, $i_0 \oplus I_p \oplus I_b$, $\cdots$, and $i_0 \oplus I_p \oplus I_x$, where $a, b, \ldots, x$, and p are distinct integers between 0 and $n-1$ (both 0 and $n-1$ inclusive). Then these $k-1$ nodes will have a common node, $i_0 \oplus I_p$, in their buddy sets. There are $n-(k-1)$ other nodes, $i_0 \oplus I_p \oplus I_k$, $0 \leq k \leq n-1$, $k \neq p, a, b, \ldots, x$, whose buddy sets contain the same node, $i_0 \oplus I_p$. So, there are $n-k+1$ ways that the buddy sets of k nodes intersect at node $i_0 \oplus I_p$. Similarly, one can show that there are $n-(k-1)$ nodes, $i_0 \oplus I_q \oplus I_k$, $0 \leq k \leq n-1$, $k \neq q, a, b, \ldots, x$, whose buddy sets intersect at node $i_0 \oplus I_q$ with those of nodes i_0, $i_0 \oplus I_q \oplus I_a$, $i_0 \oplus I_q \oplus I_b$, $i_0 \cdots$, and $i_0 \oplus I_q \oplus I_x$, where $p \neq q$. So, there are $2(n-k+1)$ ways the buddy set of k nodes overlap. □

Once the intersection of buddy sets is determined, one can calculate the average number of collisions in case of the random selection of an underloaded node in a buddy set by:

$$\begin{aligned} k_b &= \frac{\sigma\, k_s}{N} \\ \overline{c_t} &= \sum_{i=2}^{\sigma} \left[\binom{k}{i} \frac{2\, c_i}{(\sigma - k_b)^i} \left(\frac{\binom{\sigma-2}{k_b}}{\binom{\sigma}{k_b}} \right)^{i-1} \right], \end{aligned} \tag{4.20}$$

where $c_2 = \binom{n}{2}/N$, $c_j = c_{j-1} \times \frac{n-j+1}{N-j+2}$, $3 \leq j \leq \sigma$. Note that if buddy set size is greater than the hypercube's dimension, a node's buddy set must contain nodes which are two or more hops away from the node. Even in such a case, the above equation can be used to get approximate results, or the coefficients c_i's need to be adjusted to get the exact value of c_t. Again, it should be noted that in the above analysis only U–state and V–state nodes are considered. One can follow the same procedure described at the end of Section 4 to calculate the task collisions in the presence of F–state nodes.

4.5.3 Comparison of Performance of Different Location Policies

The performance of different location policies is compared in terms of the number of task collisions, number of rounds to offload overflow tasks, and communication overhead. Q_8 is chosen for this comparison.

The numbers of task collisions under the preferred list, random selection, and probing methods are plotted in Figs. 4.4 to 4.7. Generally, the number of task collisions increases with the number of overloaded nodes in all of the three location policies. When the number of overloaded nodes is less than $0.15N$, there is no significant difference in the number of collisions among the three. However, when the number of overloaded nodes is close to $0.5N$, the number of collisions of the proposed location policy with preferred lists is about 30% of that of the random selection policy and 60% of that of the probing policy. When there is no F–state node, the probing method performs as well as the preferred list method except when k approaches $N/2$. However, when the number of F–state nodes increases, the task collisions resulted from probing increase much faster than the preferred list method. As shown in Figs. 4.6 and 4.7, the probing results in more collisions than the preferred list method in all cases.

It is important to observe that use of preferred lists requires no extra state information as compared to the random selection policy. However, the number of task collisions resulting from the latter policy is more than 3 times that of the proposed policy with preferred lists. This result indicates the importance of coordinating overloaded nodes when locating underloaded nodes, which is an important issue that has been overlooked in existing local LS methods.

Task collisions under the probing policy are another interesting issue. The number of collisions is found to increase as the number of probes decreases. When only 2 (instead of 5) probes are used, this policy results in even more collisions than the random selection policy. Although the number of collisions can be reduced by increasing the number of probes, it also increases the probing delay as the number of probes increases. (Note that two messages need to be exchanged for each probe, one request and one reply.) Since an overloaded node cannot transfer an overflow task before locating an underloaded node, the probing delay will prolong the completion of the tasks to be transferred. Use of more than 5 probes is shown in [40] that the resulting probing delay will outweigh the benefit that might be gained by transferring a task from an overloaded node to an underloaded node.

The advantage of using preferred lists in general becomes more pronounced as the system gets congested. To see this tendency more clearly, suppose overloaded nodes are concentrated in an area forming a hot region. Table 4.3 shows the case when every node in a subcube is overloaded, where one to two Q_4's and one to four Q_6's are considered in 6- and 8- cubes, respectively. An interesting result can be found in Table 4.3: both the random selection and probing policies result in more collisions than the case when overloaded nodes are randomly distributed throughout the system. This indicates that both the random selection and probing policies cannot handle the congestion problem efficiently. Unsurprisingly, the location policy with preferred lists has resulted in no collision at all in these cases, because if all the nodes in a subcube are overloaded, they will transfer their overflow tasks to the underloaded nodes in another subcube, thus eliminating the possibility of task collision.

In Table 4.4, overloaded nodes are assumed to be concentrated in one to four buddy sets (overloaded buddy sets), where the address of the node whose buddy set is overloaded is given at the bottom row. Two collisions will always result if every node in a buddy set is overloaded. The first collision occurs, because the node with the overloaded buddy set will not be able to find any underloaded node to transfer an overflow task. A second collision occurs by the nodes in the overloaded buddy set as proved in Theorem 4.3. The location policy with preferred lists always results in less task collisions, except for one overloaded buddy set under the random probing policy which results in slightly less collisions. Note that in case the system is congested, Eqs. (4.15) and (4.20) need to be adjusted to calculate the number of task collisions under the random probing and selection policies, because these equations are derived under the assumption that the overloaded nodes are randomly distributed throughout the system.

The number of rounds to offload the overflow tasks is shown in Fig. 4.8. It is clear that the preferred list method requires the least number of rounds when $k > 40$. The number of tasks actually transferred in terms of k is shown in Fig. 4.9. It should be noted that in both the probing and drafting algorithms, there is no retransfer of "collided" tasks, so the number of actually transferred tasks is equal to the number of overloaded nodes, assuming that each overloaded node has only one overflow task. In all other methods the number of transferred tasks is always greater than the number of overflow tasks, but the preferred list method is still the best among these methods.

Note, however, that it is not fair to consider only the number of transferred tasks as the performance measure of a location policy, because in the probing and drafting methods many control messages need to be exchanged before transferring an overflow task. So, it is desirable to compare the performance of location policies based on a total number of messages exchanged as shown in Figs. 4.10 to 4.12. The time to transfer a task is assumed to be 2, 5, and 10 times the time of exchanging a control message between U–state and V–state nodes. It is easy to see from Fig. 4.10 that the preferred list policy results in the least number of messages. When the time to transfer a task becomes much longer than that of exchanging messages, the gap between the preferred list, probing and drafting algorithms decreases. It is found that as long as the time to transfer a task is not greater than 20 times of exchanging messages, the preferred list method will result in less communication delay than all the other methods.

Eval. methods / Overloaded nodes	simulation	approximation (median)	approximation (average)
2	0.000	0.0000	0.0000
6	0.000	6.573e-05	6.573e-05
10	0.007	0.0038	0.0038
14	0.074	0.0419	0.0419
18	0.293	0.2271	0.2297
22	0.997	0.7773	0.8709
26	2.532	1.7381	2.5484
28	3.710	2.6857	4.0582
30	5.288	4.4459	6.3214
32	7.252	6.9511	9.5599

$N = 64$

Eval. methods / Overloaded nodes	simulation	approximation (median)	approximation (average)
10	0.000	3.131e-06	3.131e-06
20	0.001	0.0004	0.0004
40	0.027	0.0318	0.0318
70	1.070	0.8793	1.0298
80	2.354	1.6096	2.3334
90	4.653	3.2155	4.7855
100	8.323	6.5668	9.1233
110	13.803	11.8384	16.3212
120	21.625	20.1099	27.7301
125	26.348	25.4312	35.5508

$N = 256$

Table 4.1: Task collisions under the random probing policy.

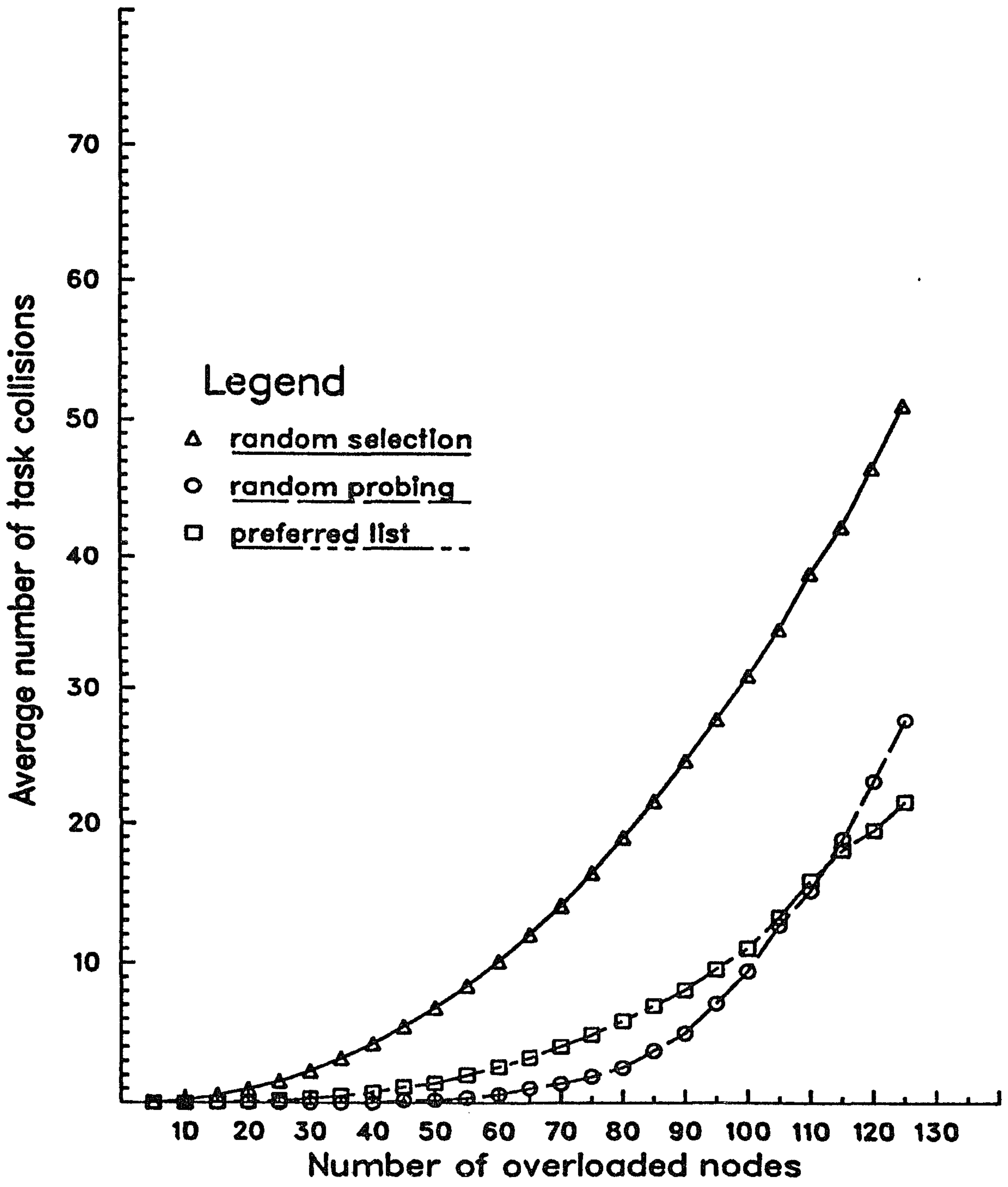


Figure 4.4: Comparison of different location policies for $N = 256$.

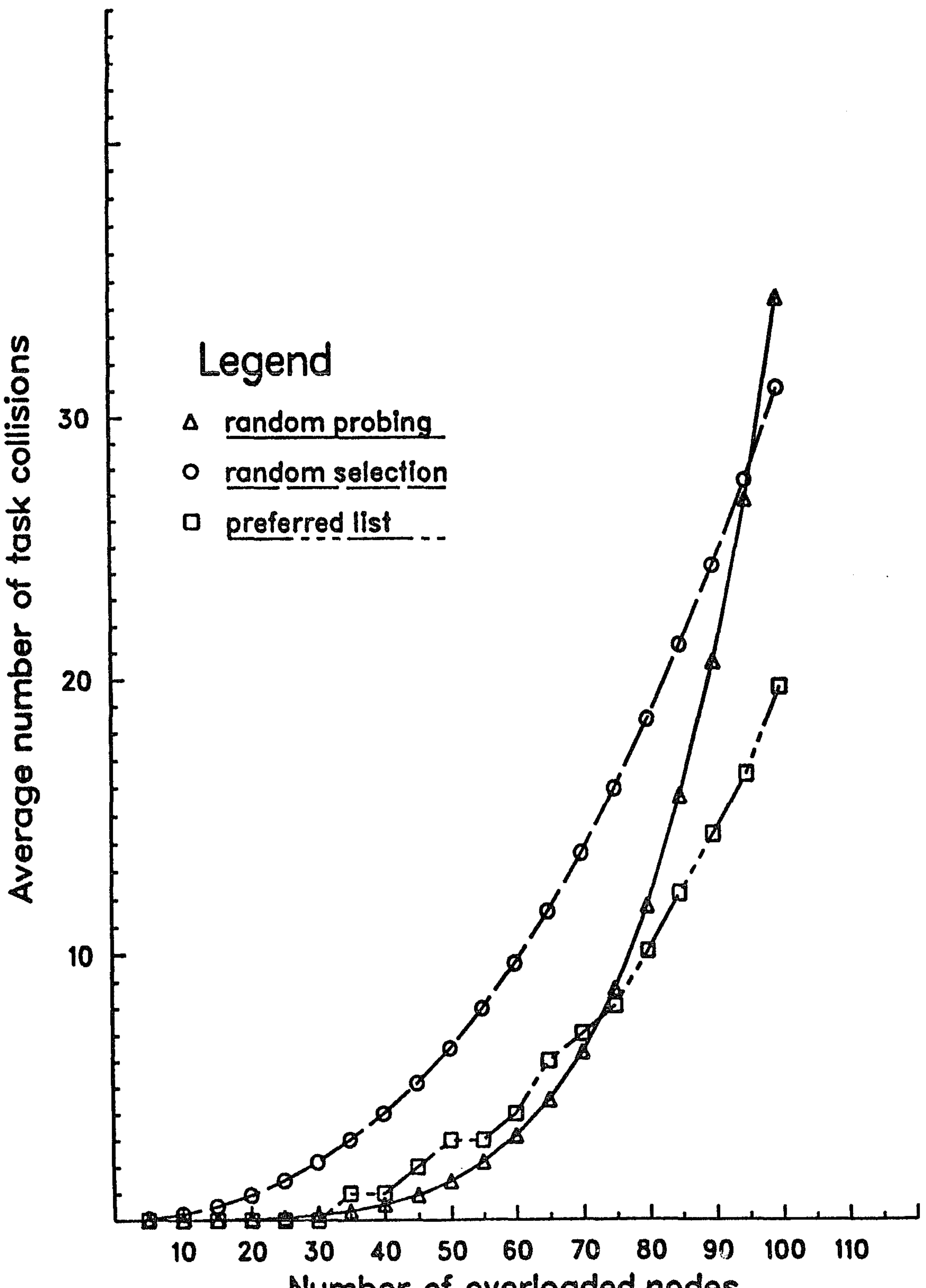


Figure 4.5: Task collisions in three location policies (fullyloaded nodes = 50).

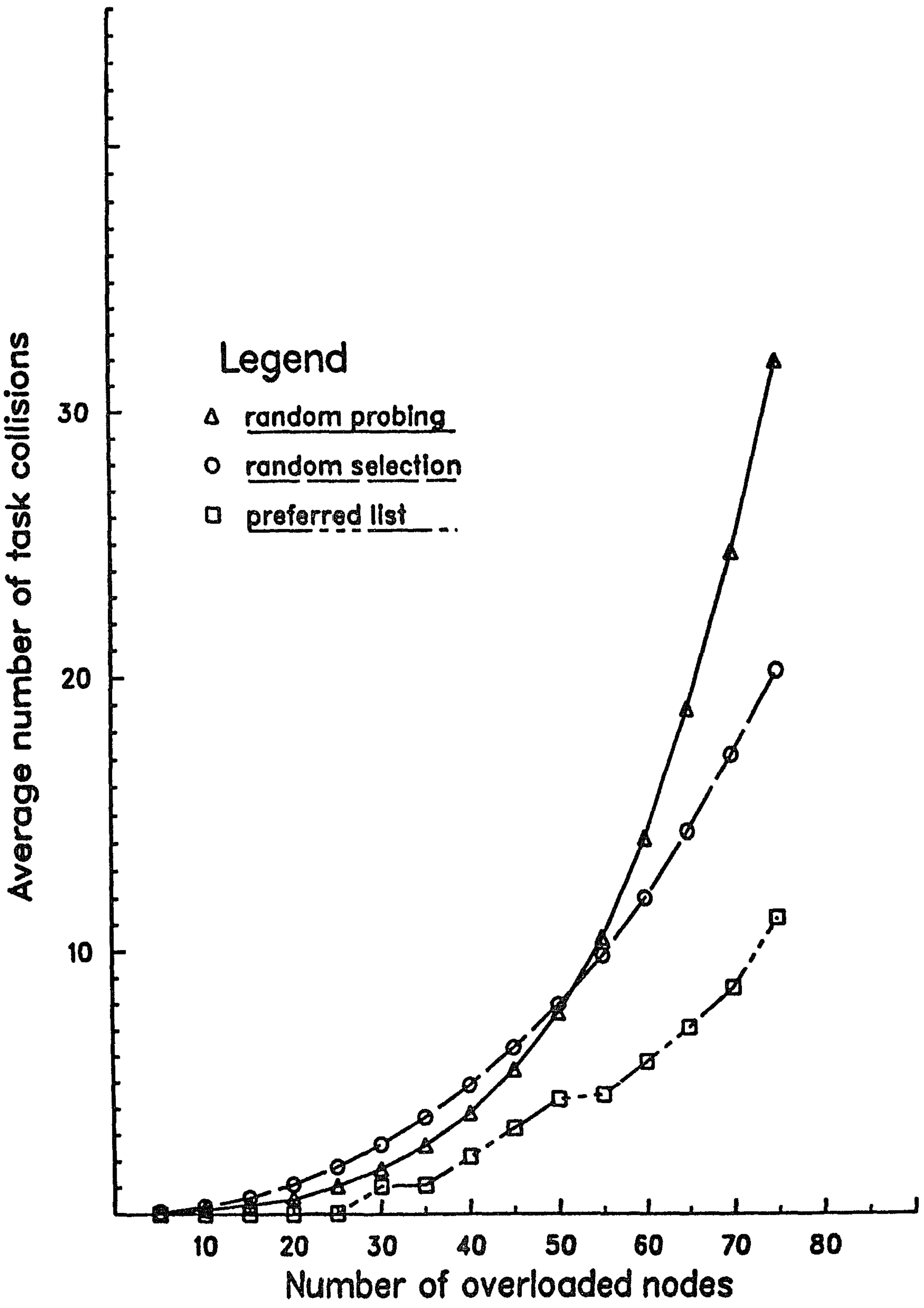


Figure 4.6: Task collisions in three location policies (fullyloaded nodes = 100).

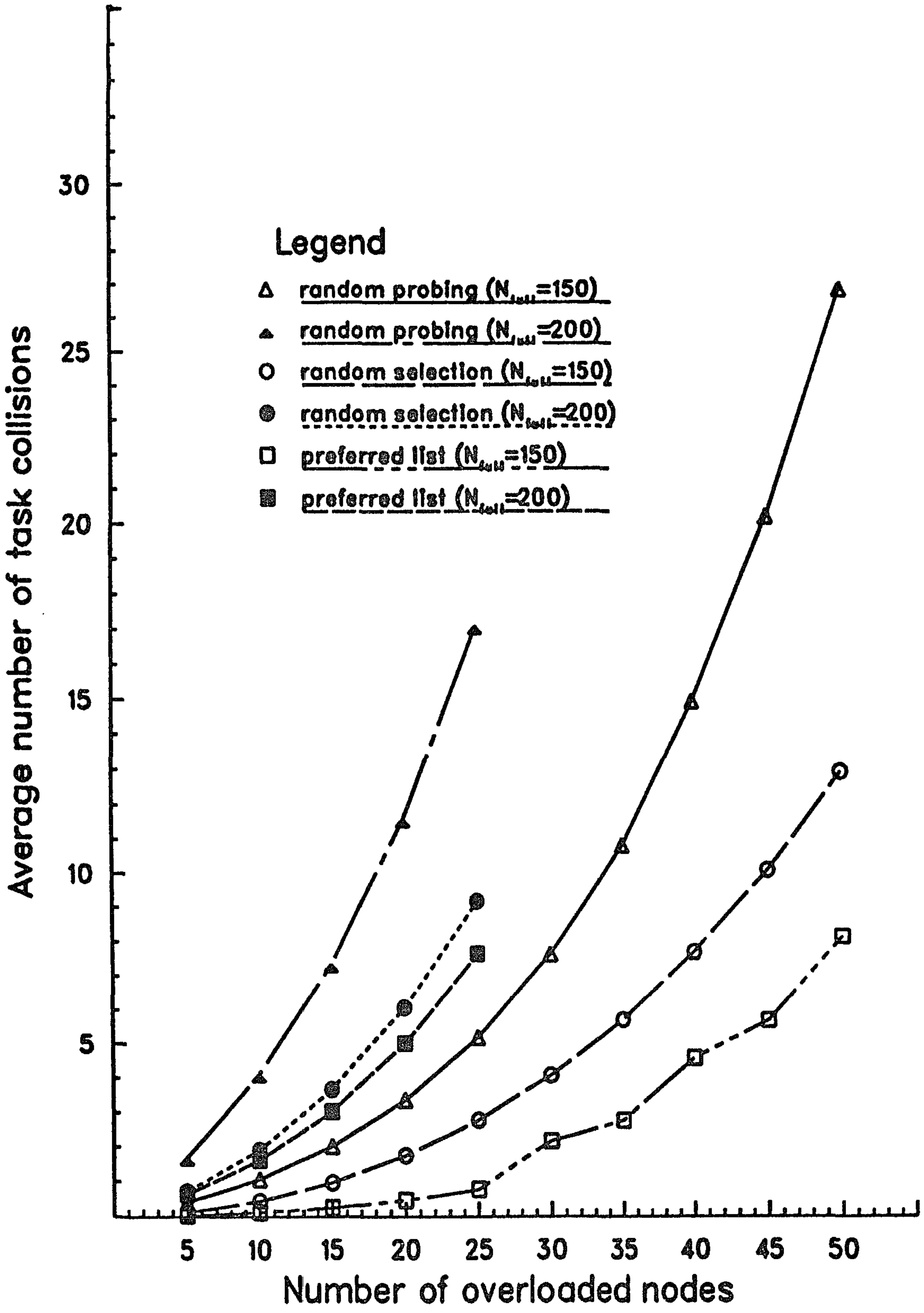


Figure 4.7: Task collisions in three location policies.

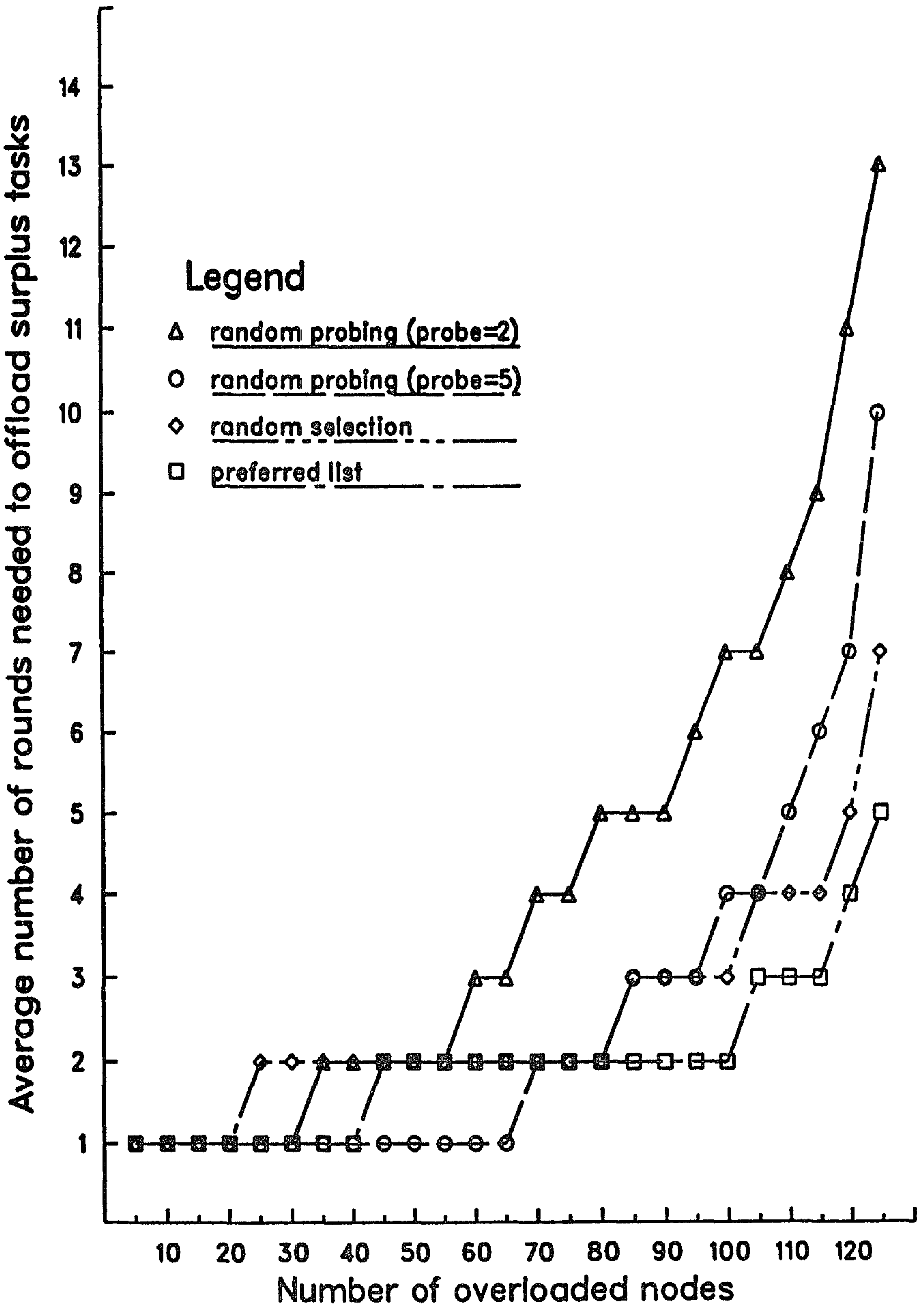


Figure 4.8: Comparison of the number of rounds needed to offload the overflow tasks in three location policies.

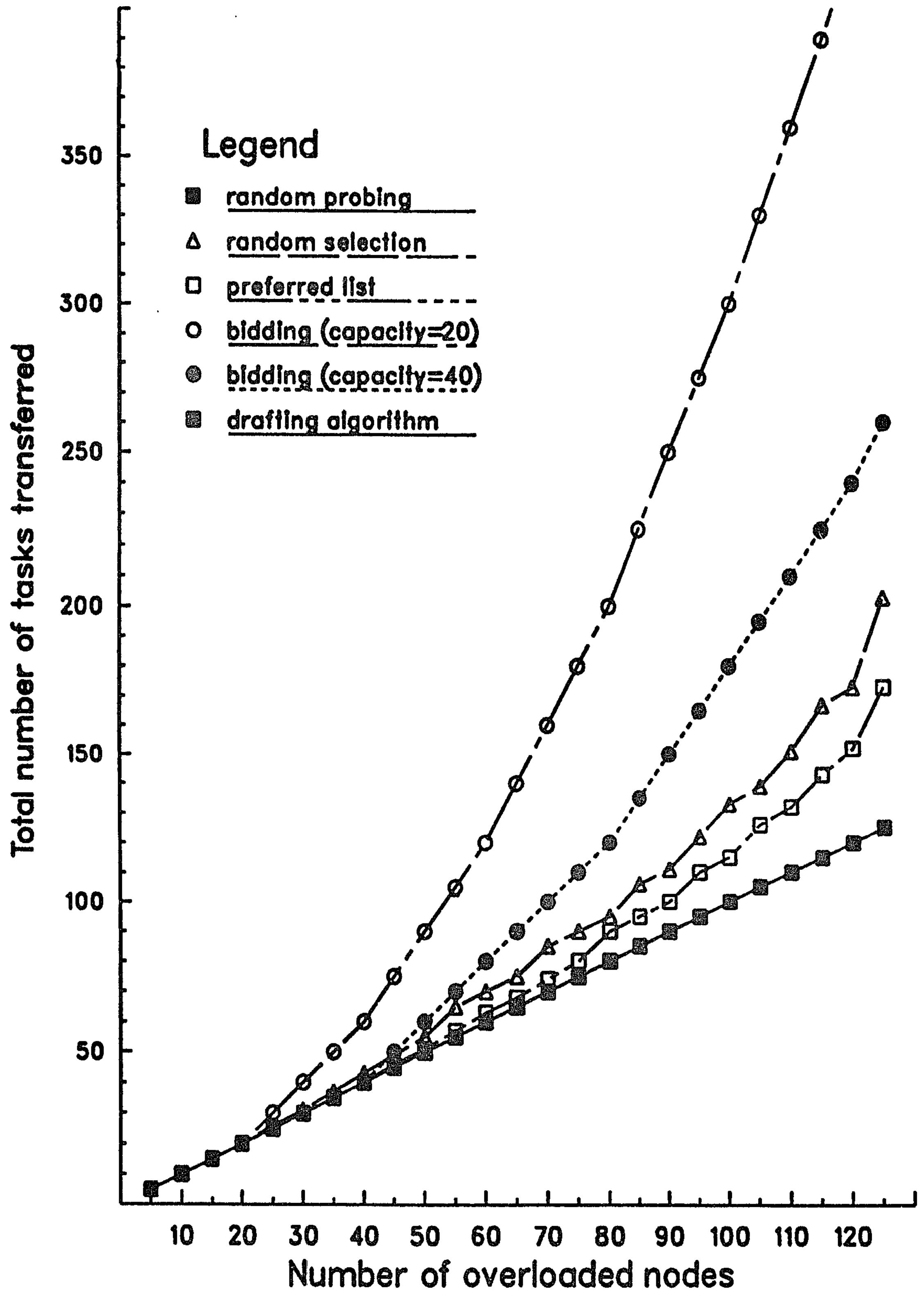


Figure 4.9: Comparison of the number of tasks transferred in different location policies.

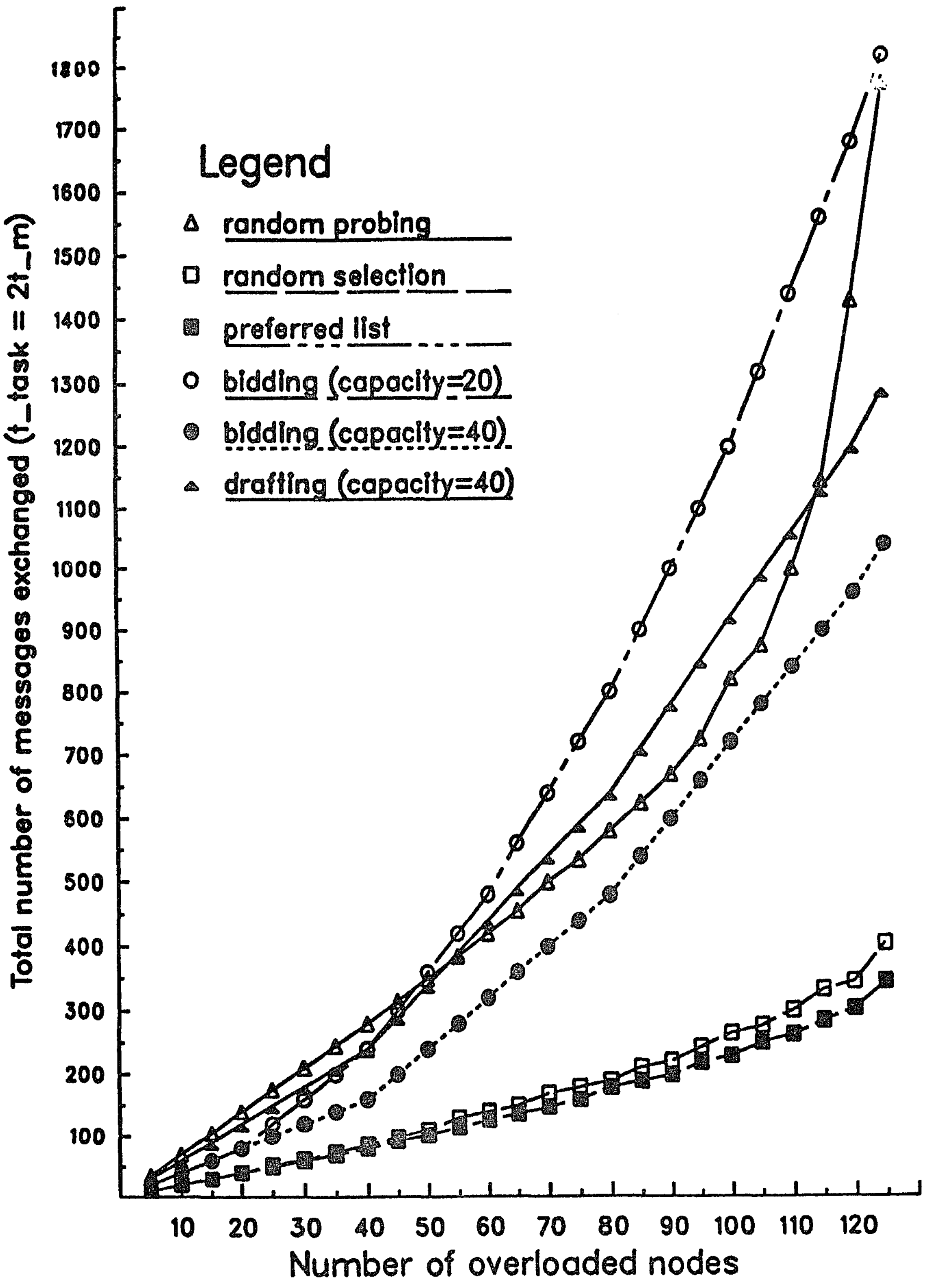


Figure 4.10: Comparison of the number of messages exchanged in different location policies.

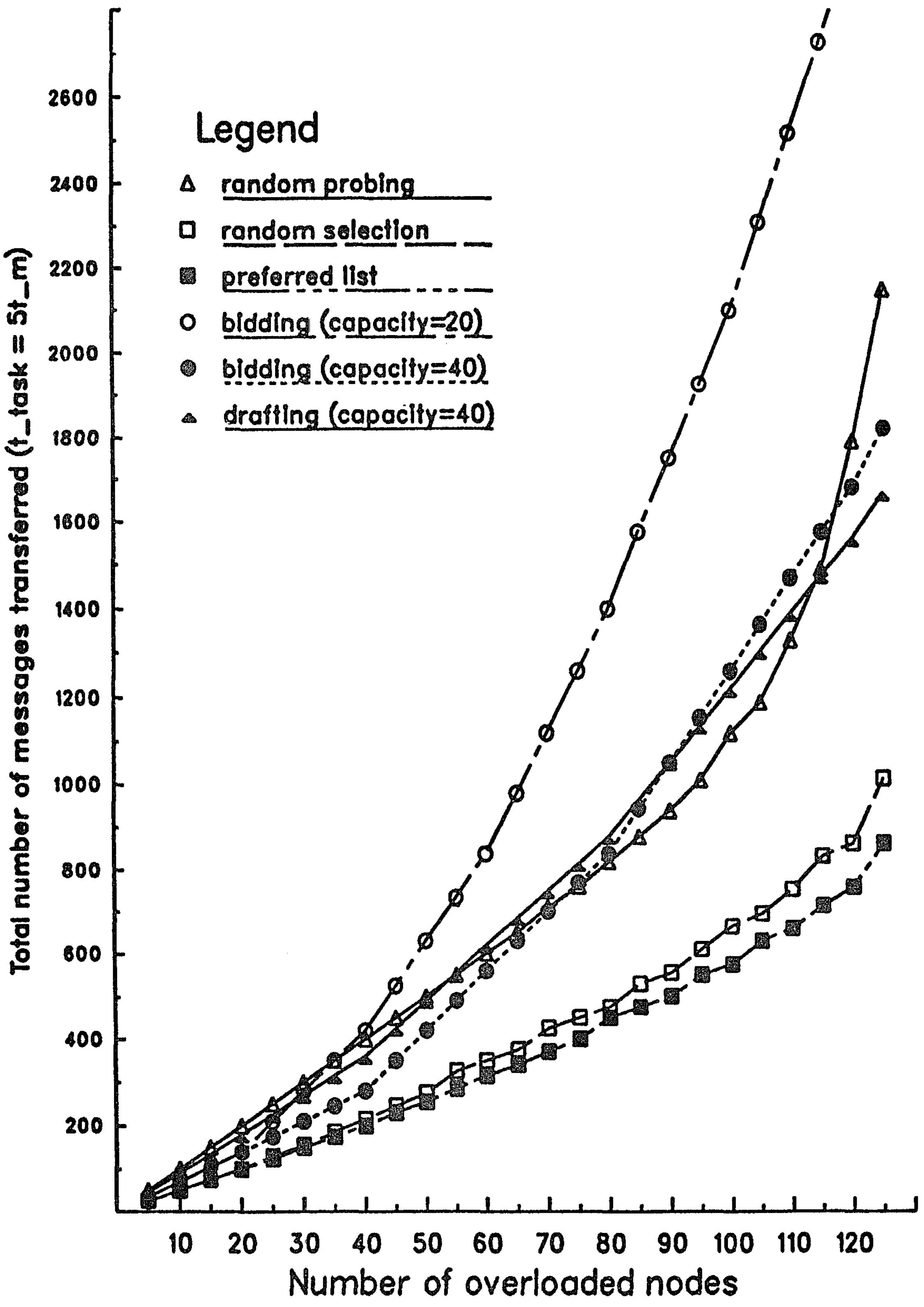


Figure 4.11: Comparison of the number of messages exchanged in different location policies.

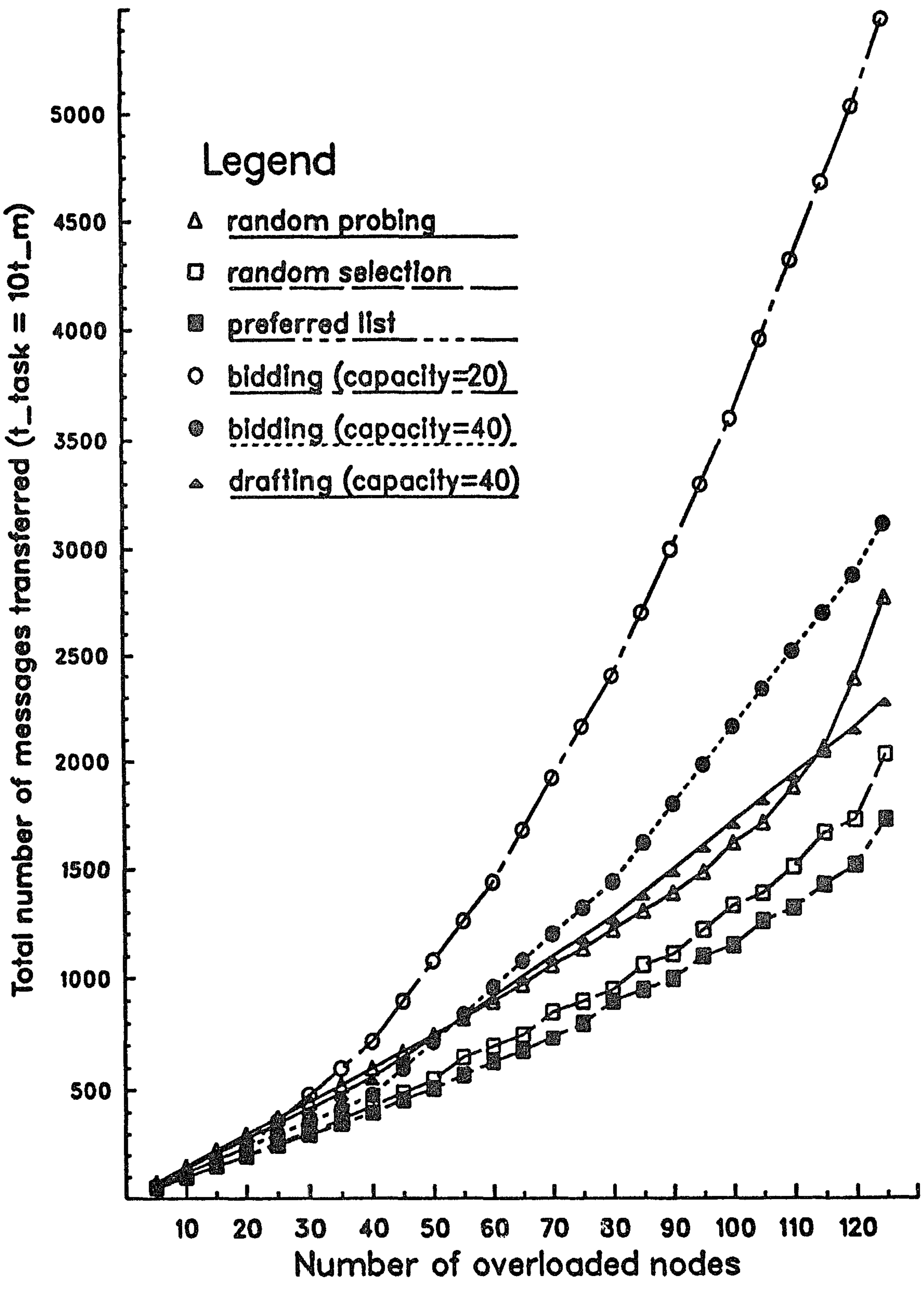


Figure 4.12: Comparison of the number of messages exchanged in different location policies.

Eval. methods / Overloaded nodes	simulation	approximation	difference (%)
2	0.021	0.0161	– 23.2
4	0.100	0.0988	– 1.1
6	0.248	0.2527	1.9
10	0.797	0.7934	– 0.4
14	1.651	1.6821	1.8
18	2.919	2.9700	1.7
22	4.677	4.7178	0.9
26	7.034	6.9956	– 0.5
30	9.944	9.8844	0.6
32	11.605	11.5857	– 0.2

N = 64, buddy set=10.

Eval. methods / Overloaded nodes	simulation	approximation	difference (%)
10	0.184	0.1809	– 0.3
20	0.793	0.7849	– 1.0
40	3.489	3.4081	– 2.3
70	11.469	11.5345	0.6
90	20.503	20.3685	– 0.6
100	25.947	26.0039	0.4
110	32.705	32.5518	– 0.5
120	40.113	40.0947	– 0.1
125	44.314	44.266	– 0.1

N = 256, buddy set=10.

Table 4.2: Task collisions under the random selection policy.

$N = 64$ Strategies	Average number of task collisions		
random selection	2.99	2.72	9.92
random probing	1.93	1.61	11.61
preferred lists	0.00	0.00	0.00
address of overloaded nodes	0 – 15	32 – 47	0 – 15, 32 – 47

$N = 256$ Strategies	Average number of task collisions		
random selection	14.8	13.28	58.36
random probing	11.57	10.13	35.41
preferred lists	0.00	0.00	0.0
address of overloaded nodes	0 – 63	128 – 191	0 – 63, 128 – 191

Table 4.3: The number of task collisions in overloaded subcubes.

$N = 64$ Strategies	Average number of task collisions		
random selection	3.65	3.32	8.67
random probing	1.54	1.48	7.31
preferred lists	2.00	2.00	4.0
overloaded buddy set(s) of	node 0	node 10	node 1 and 12

$N = 256$ Strategies	Average number of task collisions		
random selection	2.5	9.11	20.08
random probing	1.89	6.47	14.11
preferred lists	2.00	4.00	8.0
overloaded buddy set(s) of	node 0	node 1 and 63	node 0, 63, 128, 191

Table 4.4: The number of task collisions in overloaded buddy sets.

CHAPTER 5

LOAD SHARING IN HYPERCUBE MULTICOMPUTERS IN THE PRESENCE OF NODE FAILURES

5.1 Introduction

Two important fault–tolerant issues associated with the LSMSCB in hypercube multicomputers are (i) ordering fault–free nodes as preferred receivers of overflow tasks and (ii) developing a LS mechanism to handle node failures. In Chapter 4, the preferred lists and buddy sets are constructed in such a way that the coordination and congestion problems can be resolved. However, occurrence of node failures will destroy the original structure of a preferred list if faulty nodes are simply dropped from the preferred list. Thus, it is desirable to develop an algorithm to modify the preferred lists in case of node failures in order to retain the original features of LS. We will show that such a modification/adjustment is always possible regardless of the number of faulty nodes in the system.

Three algorithms are proposed to modify the preferred list to retain its original features regardless of the number of faulty nodes in the system. It is shown that either the number of adjustments or the communication overheads introduced by these algorithms is minimal.

If a node becomes faulty before completing all tasks in its queue, all of the unfinished tasks in the queue will be lost unless some fault-tolerant mechanisms are provided. Using the modified preferred lists, a simple fault-tolerant mechanism can be used to avoid/minimize tasks losses as follows. Each node is equipped with a backup queue of its most preferred node. Whenever a node fails, its most preferred node will process the unfinished tasks in the backup queue, i.e., the tasks in the failed node's queue. If this node is overloaded, it can transfer them just like those tasks arriving at the node. By using the proposed algorithms, the failed node will be replaced by a fault-free node such that the node, which was originally supposed to select the failed node as its most preferred node, will always be backed up by a fault-free node. Since the proposed adjustment algorithms are shown to induce minimal communication overhead, the probability that both a node and its most preferred node that had failed before completing the adjustment is very small, thus reducing the number of task losses significantly, as compared to those approaches without using backup queues. Two other alternative methods are simulated in which each node is backed up by two (three) nodes from its most and second (second and third) preferred nodes. The results show that the simple mechanism with one backup queue performs as well as those more complex alternatives, insofar as the number of faulty nodes is less than 40% of the total number of nodes.

The adjustment algorithms and fault-tolerant mechanisms are discussed in Section 5.2. The performance of the proposed approach is evaluated via modeling and simulation in Section 5.3.

5.2 LSMSCB in the Case of Node Failures

Node failures are detected by the other nodes through communication time-outs. Upon detection of a node failure, how to modify the preferred lists for the purpose of LS is the subject of this section. Implementation of backup queues will be treated in Section 5.3.

5.2.1 Modification of Preferred Lists

Before delving into the adjustment algorithms, it is convenient to introduce two operators, $\rightarrow$ and $\leftarrow$, which are used to describe the relation between nodes in a preferred list. Note that $N_{i \rightarrow j}$ is the j^{th} preferred node of N_i, while $N_{i \leftarrow j}$ is the node that selects N_i as its j^{th} preferred node. When two or more operators are combined in an expression, the interpretation applies from left to right. For example, $N_{i \rightarrow j \rightarrow k}$ is the k^{th} preferred node of $N_{i \rightarrow j}$, and $N_{i \rightarrow j \leftarrow k}$ is the node that selects $N_{i \rightarrow j}$ as its k^{th} preferred node. Similarly, $N_{i \rightarrow j \rightarrow k \rightarrow \ell}$ is the ℓ^{th} preferred node of $N_{i \rightarrow j \rightarrow k}$ and $N_{i \rightarrow j \rightarrow k \leftarrow \ell}$ is the node that selects $N_{i \rightarrow j \rightarrow k}$ as its ℓ^{th} preferred node. The two operators can be combined in any order; for example, $N_{i \leftarrow j \rightarrow k}$ is the k^{th} preferred node of the node which selects N_i as its j^{th} preferred node and $N_{i \leftarrow j \rightarrow k \leftarrow \ell}$ is the node that selects $N_{i \leftarrow j \rightarrow k}$ as its ℓ^{th} preferred node, and so on. For the preferred lists in Fig. 2.2, examples of using the two operators are $N_{0 \rightarrow 1 \rightarrow 2} = N_3$, $N_{0 \rightarrow 1 \leftarrow 3} = N_5$, $N_{0 \leftarrow 2 \rightarrow 3} = N_6$, and $N_{1 \leftarrow 2 \leftarrow 3} = N_7$.

If faulty nodes are simply dropped from the preferred lists, some nodes will be selected by more than one node as the ℓ^{th} preferred nodes for some $1 \leq \ell \leq \sigma$. For example, suppose N_0 is faulty, then the preferred lists of the nodes in $S_{N_0}^{-1} = \{N_{0 \leftarrow 1}, N_{0 \leftarrow 2}, \ldots, N_{0 \leftarrow \sigma}$ $\} = \{N_1, N_2, N_4, N_8, N_6, N_{10}$ $\}$ will be changed as follows. The $(k+j)^{th}$ preferred node of $N_{i \leftarrow j}$ will become the $(k+j-1)^{th}$ preferred node for $j = 1, \ldots, \sigma$ and $k = 1, \ldots, \sigma - j$. However, according to Theorem 4.1, the $(k+j)^{th}$ preferred node of $N_{i \leftarrow j}$ (i.e., $N_{i \leftarrow j \rightarrow k+j}$) is already assigned as the $(k+j-1)^{th}$ preferred node of another node $N_{i \leftarrow j \rightarrow k+j \leftarrow k+j-1}$, so $N_{i \leftarrow j \rightarrow k+j}$ will be selected by two nodes as the $(k+j-1)^{th}$ preferred node when the faulty node N_0 is dropped from the preferred list of $N_{i \leftarrow j}$.

Generally, if there are f faulty nodes, none of which are located in the same buddy set, then there will be $f \times k$ nodes to be selected as the k^{th} preferred node of two other nodes. However, if some faulty nodes are located in the same buddy set, the number of nodes that will be selected by more than two nodes is very difficult to derive. Table 5.1

lists the number of nodes that are selected by more than two nodes in an 8-cube system.

In case of node failures, some nodes are likely to be selected by more than one node at a time and the properties stated in Theorem 4.1 and 4.2 will no longer hold. Moreover, in the proposed fault-tolerant backup queue mechanism of Section 4, the backup node is assigned based on the corresponding preferred list. If a node were selected by more than one node as their most preferred node, the failure of this node will affect more than one node. Thus, it is desirable to adjust the preferred lists dynamically whenever a node becomes faulty such that Theorem 4.1 and 4.2 will hold even in case of node failures. The following theorem states that such an adjustment is always possible but not unique.

Theorem 5.1 *Regardless of the number of faulty nodes in a system, there always exists an algorithm to adjust the preferred lists such that Theorem 4.1 and 4.2 will always hold. Moreover, such an adjusting algorithm is not unique.*

Proof: Let S_g be the ordered set of fault-free nodes in the system. Suppose $|S_g| = x$, and let N_i be the i^{th} node in S_g. The following preference assignment in S_g will not violate the conditions of Theorem 4.1 and 4.2. For example, N_1 can choose any other node (in $x - 1$ ways) as its most preferred node, N_2 can choose any node except itself and the one just picked by N_1 as its most preferred node (in $x - 2$ ways), and so on. Thus, there are $(x - 1)!$ ways to assign the most preferred node to each of the nodes in S_g. The assignment of the second preferred node for each node in S_g without violating the conditions of Theorem 4.1 and 4.2 can be obtained by letting N_i pick N_{i+1}'s most preferred node for $i = 1, \ldots, x - 1$ while letting N_x pick N_1's most preferred node. The rest of the preferred lists can be constructed in a similar way. Thus, there are at least $(x - 1)!$ ways to modify a preferred list for the nodes in S_g without violating the conditions of Theorem 4.1 and 4.2.
□

When N_i becomes faulty, the preferred list of each node in $S_{N_i}^{-1}$ needs to be adjusted. Since adjusting a preferred list will introduce computation and communication overheads, it is desirable to develop an algorithm which requires a minimal number of adjustments in case of node failures (Section 3.2). Moreover, in the fault-tolerant backup queue approach, the most preferred node should be located as close to the failed node as possible to reduce the communication overhead for maintaining/updating the backup queue (Section 3.3).

5.2.2 The Number of Adjustments

An adjustment algorithm is said to be 'optimal' if the number of nodes needed to be adjusted is minimal.

Theorem 5.2 *When N_i is faulty, the minimal adjustment to the preferred list of each node in $S_{N_i}^{-1}$ is to change the k^{th} preferred node of <u>either</u> one node if $N_{i\to k} \neq N_{i\leftarrow k}$ <u>or</u> two nodes if $N_{i\to k} = N_{i\leftarrow k}$ for $k = 1, \ldots, \sigma$ without violating the conditions of Theorems 4.1 and 4.2.*

Proof: Since N_i is the k^{th} preferred node of $N_{i\leftarrow k}$, it needs to be replaced by a fault-free node. Note that the k^{th} preferred node of N_i (i.e., $N_{i\to k}$) will not be the k^{th} preferred node of any other node according to Theorem 4.1 because N_i is faulty. If $N_{i\to k} \neq N_{i\leftarrow k}$, one can simply substitute $N_{i\to k}$ for N_i as the k^{th} preferred node of $N_{i\leftarrow k}$. This adjustment will satisfy the conditions of Theorems 4.1 and 4.2.

However, in most cases $N_{i\to k} = N_{i\leftarrow k}, \forall k$. It can be shown that there always exist a pair of nodes (say N_x–N_y), such that changing the k^{th} preferred node of both $N_{i\to k}$ and N_x will satisfy the conditions of Theorem 4.1 and 4.2. For notational convenience, let N_x and N_y be two distinct nodes, such that $N_y = N_{x\to k}$, $N_{i\leftarrow k} \notin S_{N_x}$, and $N_y \notin S_{N_{i\leftarrow k}}$. Then the following adjustments:

Replace N_i with N_y as the k^{th} preferred node of $N_{i\leftarrow k}$

Replace N_y with $N_{i\leftarrow k}$ as the k^{th} preferred node of N_x

will satisfy the conditions of Theorem 4.1 and 4.2 for the following reasons. Theorem 4.1 is satisfied by assuring $N_{i\leftarrow k} \notin S_{N_x}$ and $N_y \notin S_{N_{i\leftarrow k}}$. Before making this adjustment, $N_{i\leftarrow k}$ is the kth preferred node of N_i only and thus will not be selected as the kth preferred node by any other node when N_i becomes faulty. So, $N_{i \leftarrow k}$ (N_y) will be selected as the kth preferred node by N_x ($N_{i \leftarrow k}$) only after making the adjustment, thus satisfying Theorem 4.2.

As long as the size of a buddy set is less than a half of the system, the (N_x–N_y) pair always exists in the system. The restriction $N_y \notin S_{N_{i \leftarrow k}}$ implies that N_y must be at least two hops away from $N_{i \leftarrow k}$, and the relation $N_y = N_{x \rightarrow k}$ implies that N_x must be at least three hops away from $N_{i \leftarrow k}$. □

The pair N_x–N_y can be found systematically for each node in $S_{N_i}^{-1}$ as follows. Since N_y will replace N_i as $N_{i \leftarrow k}$'s kth preferred node, it must not be in $N_{i \leftarrow k}$'s buddy set. So, one can search $N_{i \leftarrow k}$'s preferred list, starting from the first node which is <u>outside</u> of its buddy set. (Note that N_i's buddy set is formed with the first σ nodes in N_i's preferred list.) If this node is two (three) hops away from $N_{i \leftarrow k}$, it can be found by $N_{i \leftarrow k} \oplus I_p \oplus I_q$ ($N_{i \leftarrow k} \oplus I_p \oplus I_q \oplus I_r$), where $p = 1, 2, \ldots, n-1, 0,\ \ q = p+1, 2, \ldots, n-1$, and $r = q+1, 2, \ldots, n-1$. The next step is to ensure that $N_{i \rightarrow k}$ is not in N_x's buddy set. This can be accomplished by keeping N_x farther away from $N_{i \rightarrow k}$. Without loss of generality, one can consider the case of $k \leq n$, and assume that the buddy set of a node N_k contains all nodes one hop away from N_k and some of the nodes which are two hops away from N_k. The replacement of faulty nodes can be found by using Algorithm 5.1.

<u>**Algorithm 5.1**</u>: Minimal adjustment of a preferred list.

```
for  $k = 1$ to $\sigma$ do
    1. Find $N_y = N_{i \leftarrow k} \oplus I_p \oplus I_q$, such that $p, q \neq k-1$ and $N_y \notin S_{N_{i \leftarrow k}}$
    2. Replace $N_i$ by $N_y$ as $N_{i \leftarrow k}$'s kth preferred node.
    3. Find $N_x = N_y \oplus I_{k-1}$
    4. Replace $N_y$ by $N_{i \rightarrow k}$ as $N_x$'s kth preferred node.
end_do
```

Theorem 5.3 *The preferred list resulting from the adjustment Algorithm 5.1 will satisfy the conditions of Theorems 4.1 and 4.2 in the presence of faulty nodes.*

Proof: According to the definition of a preferred list, $N_{i \rightarrow k} = N_i \oplus I_{k-1}$ and $N_y = N_{i \leftarrow k} \oplus I_p \oplus I_q = N_i \oplus I_{k-1} \oplus I_p \oplus I_q$. Then $N_x = N_{y \leftarrow k} = N_y \oplus I_{k-1} = N_i \oplus I_p \oplus I_q$ is farther away from $N_{i \leftarrow k}$ if $p, q \neq k-1$ than before the adjustment. Since a preferred list is generated according to the distance between nodes, if N_y is not in $N_{i \leftarrow k}$'s buddy set, then N_x, which is farther away from $N_{i \leftarrow k}$ than before the adjustment, will not be in its buddy set either. So, the pair N_x–N_y will satisfy the conditions of Theorems 4.1 and 4.2. □

Although Algorithm 5.1 can modify the preferred lists by using a minimal number of adjustments, the distance between a node and the nodes in its buddy set is found to increase significantly after these adjustments for the following reason. Whenever N_i becomes faulty, the nodes in $S_{N_i}^{-1}$ must adjust their preferred lists. In each of these adjustments a pair of nodes N_x–N_y need to be selected. According to Algorithm 5.1, every node in $S_{N_i}^{-1}$ will select the first fault-free node as N_y in $\overline{S_{N_{i \leftarrow k}}}$. Suppose $N_y = N_i \oplus I_{k-1} \oplus I_p \oplus I_q$, then $N_x = N_{y \leftarrow k} = N_y \oplus I_{k-1} = N_i \oplus I_p \oplus I_q$. So, $N_x = N_i \oplus I_p \oplus I_q$ is a common element of all node pairs used for adjusting preferred lists. In other words, after completing the adjustments for every node in $S_{N_i}^{-1}$, the nodes in node $N_i \oplus I_p \oplus I_q$'s buddy set will be at least three hops away from this node, making the average distance greater than 2, while the average distance is around 1 in all other nodes' buddy sets.

To alleviate this problem, Algorithm 5.1 is modified in such a way that a different pair N_x–N_y of nodes is selected for each node in $S_{N_i}^{-1}$. In such a case, after completing the adjustments for all nodes in $S_{N_i}^{-1}$, the average distance between a node and the nodes in its buddy set will become smaller than the average distance resulting from Algorithm 5.1. From the definition of a preferred list, the nodes which are two hops away from N_i are ordered according to the operation of $N_i \oplus I_j \oplus I_k$ ($j = 1, \ldots, n-1, 0$, and $j+1 \leq k \leq n-1$). For convenience, the nodes with the same j in the above operation are grouped as parcel

j, denoted as Ψ_j^i (parcel j of node i). The k^{th} node in Ψ_j^i can be obtained by $N_i \oplus I_j \oplus I_k$. For the example preferred lists in Fig. 2.2, $\Psi_0^1 = \{N_1, N_2, N_4, N_8\}$, $\Psi_1^0 = \{N_6, N_{10}\}$ and $\Psi_2^3 = \{N_{15}\}$.

Algorithm 5.2: Adjustment of a preferred list with reduced average inter-node distances in a buddy set:

```
For  $k = 1$ to $\sigma$ do
    1. Find the first parcel $j$ of $N_{i \leftarrow k}$ for $j \neq k-1$.
    2. If  $j = 1$ then
            if  $j + k \leq n - 1$ then $N_y = N_{i \leftarrow k} \oplus I_j \oplus I_{j+k}$
            else $j \longleftarrow j + 1$ goto Step 2.
       else $p = k - \sum_{\ell=1}^{\ell=j} |\Psi_\ell^i|$
            if  $p \geq 0$ then $N_y = N_{i \leftarrow k} \oplus I_j \oplus I_{j+p}$
            else $N_y = N_{i \leftarrow k} \oplus I_j \oplus I_{j+1}$
    3. Replace $N_i$ by $N_y$ as $N_{i \leftarrow k}$'s $k^{th}$ preferred node.
    4. Find $N_x = N_y \oplus I_k$.
    5. Replace $N_y$ by $N_{i \rightarrow k}$ as $N_x$'s $k^{th}$ preferred node.
end_do
```

The difference between Algorithm 5.1 and Algorithm 5.2 can best be explained by the following example. Let N_0 be the only faulty node in a Q_4 and assume $\sigma = 4$. Then $S_{N_0} = \{N_1, N_2, N_4, N_8\}$. From Algorithm 5.1, the pair N_x–N_y for each node in S_{N_0} will be N_6–N_7, N_6–N_4, N_6–N_2, N_6–N_{14}. So, after completing the adjustments, N_6's buddy set will become $\{N_1, N_2, N_4, N_8\}$ where N_1 and N_8 are three hops away from N_6. However, from Algorithm 5.2, the pair N_x–N_y for each node in S_{N_0} will be N_6–N_7, N_{12}–N_{14}, N_{10}–N_{14}, and N_3–N_{11}. In this case, N_6's buddy set will become $\{N_7, N_4, N_2, N_{14}\}$. It is easy to see that the average distance between N_6 and the nodes in its buddy set is reduced by using Algorithm 5.2.

5.2.3 Minimization of Average Inter-node Distance in a Buddy Set

Although the number of adjustments in preferred lists is minimized by Algorithm 5.2, the distance between a node and its most preferred node is not necessarily minimal. When N_i is faulty, according to Algorithm 5.2, the k^{th} preferred node of $N_{i \leftarrow k}$ (i.e., N_i) is replaced by N_y which is two or more hops away from $N_{i \leftarrow k}$. Moreover, the k^{th} preferred node of N_x (i.e., N_y) is replaced by $N_{i \rightarrow k}$ which is at least three hops away from N_x. For example, when $k = 1$, the most preferred nodes of $N_{i \leftarrow 1}$ and N_x will be two and three hops away from $N_{i \leftarrow 1}$, respectively.

Since each node N_i needs to be aware of the tasks arriving at the node(s) to be backed up, the longer distance between N_i and the node(s) to be backed up means larger communication delays between them. So, we have to minimize the distance between the nodes and their replacement nodes.

First, we want to find the condition under which the most preferred node of a node can always be located one hop away irrespective of the number of faulty nodes in the hypercube. We will then develop an algorithm which minimizes the average distance between a node and the nodes in its buddy set.

Lemma 5.1 *If there are an odd number of faulty nodes in a hypercube, the only way to locate every fault-free node's most preferred node within one hop of itself is to find a cycle with an odd number of fault-free nodes.*

Proof: Suppose there are m (an odd number) fault-free nodes in a hypercube. Each node's most preferred node can be found within one hop of itself as follows. First, one can find a path with an even number of nodes, say, $N_0, N_1, \ldots, N_{k-1}$ (k is an even number). Then N_i and N_{i+1} can be chosen as the most preferred node to each other for $i = 0, \ldots, i - 2$.

For the remaining odd number of fault-free nodes, the only way to assign each node the most preferred node within one hop is to find a cycle formed by these nodes. Let

$N_k, N_{k+1}, \ldots, N_{k+\ell-1}$ be the nodes in this cycle. Then one can assign N_{k+i} as the most preferred node of N_{k+i+1} and $N_{k+\ell}$ will be the most preferred node of N_k for $i = 0, \ldots, \ell-1$. However, if one cannot find a cycle in the remaining nodes, it is impossible to assign a node the most preferred node within one hop. □

Theorem 5.4 *If there are an odd number of faulty nodes in a hypercube, it is impossible to assign every fault-free node the most preferred node within one hop.*

Proof: From Lemma 5.1 the only way to assign every fault-free node the most preferred node within one hop is to find cycles with odd numbers of fault-free nodes. This theorem is proved by showing that such cycles do not exist in hypercubes.

Let the sequence $N_0, N_1, \ldots, N_k$ be the nodes in a cycle. N_i and N_{i+1} are adjacent to each other for $i = 0, \ldots, k-1$. Using the $\oplus$ operation for $0 \leq i \leq k-1$, this sequence of nodes is represented as follows. $N_{i+1} = N_i \oplus I_p$, and $N_{i+2} = N_{i+1} \oplus I_q = N_i \oplus I_p \oplus I_q$ for some $0 \leq p, q \leq n-1$. Then it is easy to show that N_{i+1} is one hop away from N_0, and N_{i+2} is either two hops away from N_0 if $p \neq q$ or equal to N_0 if $p = q$. Repeating this procedure, $N_{i+3} = N_0 \oplus I_p \oplus I_q \oplus I_r$ is either one hop away from N_0 if any two of p, q, r are equal, or three hops away from N_0 if p, q, r are all distinct integers. Generally, one can show that N_{i+j} will be an odd (even) number of hops away from N_0 if j is odd (even).

In order to form a cycle, the start and end node must be one hop away from each other, so j can only be an odd number making the total number of nodes in a cycle even.

□

Every node is guaranteed to have the most preferred node within one hop only when there are an even number of fault-free nodes that form a Q_k, $k \geq 1$. We prove this formally in the following lemma.

Lemma 5.2 *The most preferred nodes of all nodes in a subcube form another subcube of the same dimension.*

Proof: According to the definition of a preferred list, the most preferred nodes of all nodes in a Q_k can be found by $I_0 \oplus$ every node's address of the Q_k. Since the $\oplus$ operation with I_0 complements the digit at position '0' and does not change any other digits, the result of the above operation gives another subcube of the same dimension. □

Theorem 5.5 *If there are an even number of faulty nodes in a hypercube and all faulty nodes form at least one Q_k, $k \geq 1$, it is always possible to assign every fault-free node the most preferred node within one hop.*

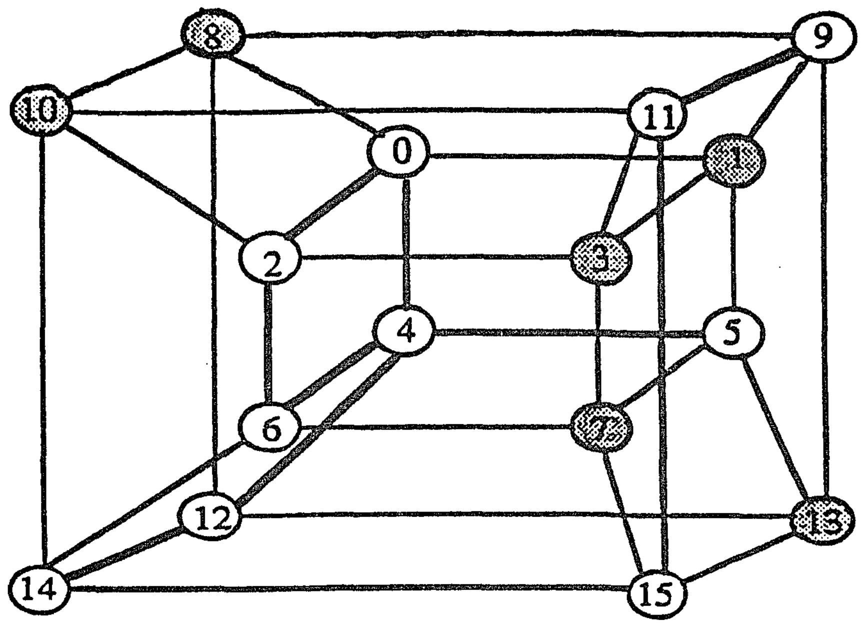
Proof: According to Lemma 5.2, the most preferred nodes of nodes in a subcube also form a subcube of the same dimension. When the faulty nodes form Q_k's, $k \geq 1$, one can always find an even number of fault-free nodes in a subcube such that these nodes can be paired to be the most preferred node to each other. □

If there are an even number of faulty nodes and some of these nodes do not form a Q_k, $k \geq 1$, it may or may not be able to locate the most preferred node within one hop for every fault-free node. For example, as shown in Fig. 5.1(a) it is impossible to find a path with an even number of nodes among the fault-free nodes, so at least two nodes cannot be assigned the most preferred node within one hop. However, in Fig. 5.1(b) one can find two paths connected with an even number of nodes, i.e., path N_1–N_5 and path N_9, N_{11}, ..., N_6, and N_2.

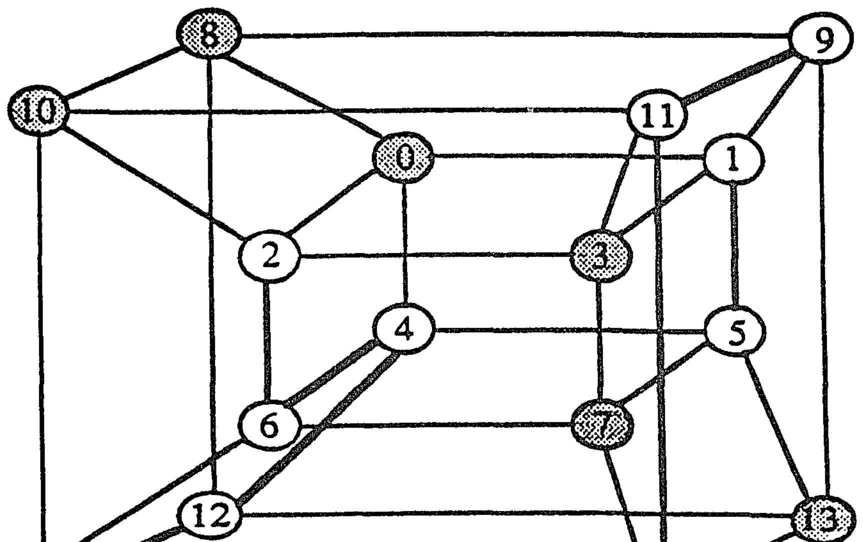
The following algorithm will adjust the preferred list of each fault-free node such that the most preferred node is located one hop away from each fault-free node if Theorem 5.5 holds; otherwise, one node will be assigned the most preferred node within one hop and the other node within two hops.

Algorithm 5.3: Adjustment of a preferred list with minimal average inter-node distance:

```
For  $k = 1$ to $\lfloor \sigma/2 \rfloor$ do
    If  $N_{i \leftarrow k}$ is not faulty then
```



(a).



(b).

Figure 5.1: Example of assigning most preferred nodes in the presence of faulty nodes

```
            Replace $N_i$ with $N_{i\leftarrow k\rightarrow n-k}$ as the $k^{th}$ preferred nodes of $N_{i\leftarrow k}$
            Replace $N_{i\leftarrow k\rightarrow n-k}$ with $N_{i\rightarrow k}$ as the $k^{th}$ preferred node of $N_{i\leftarrow k\rightarrow n-k\leftarrow k}$
            Use the same procedure in Algorithm 5.2 to find a node to replace
                $N_{i\leftarrow k\rightarrow n-k\leftarrow k}$ as the $(n-k)^{th}$ preferred node of $N_{i\leftarrow k}$
        else    no adjustment
end_do
for $k = \lfloor \sigma/2 \rfloor + 1$ to $\sigma$ do
    Same as Algorithm 5.2
end_do
```

Note that in Algorithm 5.3, the distance between a node and its most preferred node is always one for $N_{i\leftarrow k}$, $k < \lfloor \sigma/2 \rfloor$, but it requires four adjustments (if $N_{i\leftarrow k}$ is not faulty). This algorithm can be run concurrently on every node in $S^{-1}_{N_i}$, as long as N_i's failure is detected by the nodes in $S^{-1}_{N_i}$. When there are more than one faulty node, Algorithm 5.2 and 5.3 can be used sequentially to make the adjustments. An example of adjusting the preferred list in an 8-cube is shown in Fig. 5.2.

The number of adjustments, and the average distance between a node and its most preferred node resulting from these algorithms are compared in Table 5.2. It is shown that Algorithm 5.1 results in the largest distance among the three schemes while Algorithm 5.3 results in minimal distance between a node and its most preferred node. If the number of adjustments is the main concern, Algorithm 5.2 should be used because it results in a smaller average distance than Algorithm 5.1 while minimizing the number of adjustments.

5.3 Implementation and Analysis of Backup Queues

Based on the proposed adjustment algorithms, a simple fault-tolerant mechanism can be implemented to reduce the number of task losses when a node fails. Each node maintains two task queues, one for its own arrivals (EAQ) and the other for the arrivals from its most preferred node (BKQ). The BKQ is updated upon arrival/completion of each task at a node's most preferred node.

Assuming that N_i fails at time t, node $N_{i\rightarrow 1}$ will accept all unfinished tasks in its BKQ (as bursty arrivals). Based on the LS method in [25], it will process all of these tasks if it can meet their deadlines; otherwise, it will transfer some or all of these tasks to other underloaded nodes in its buddy set. The most important issue in this approach is to adjust the preferred list of $N_{i\leftarrow 1}$ and update its BKQ with the newly–assigned, most preferred node.

Since each node in $S_{N_i}^{-1}$ can execute the adjustment algorithm concurrently with other nodes in $S_{N_i}^{-1}$, the communication delay, T_{adjust}, associated with updating the BKQ of node $N_{i\leftarrow 1}$ and the newly–assigned, most preferred node N_x, can be derived as $T_{adjust} = (k_1 + k_2) \times T_t + T_c$, where k_1 and k_2 are the numbers of tasks in the EAQ of $N_{i\leftarrow 1}$ and $N_{x\rightarrow 1}$, respectively, T_t is the task transfer time, and T_c is the communication time between $N_{i\rightarrow 1}$ and N_x, or between $N_{x\leftarrow 1}$ and $N_{i\rightarrow 1}$ required to set up the updating procedure. If $N_{i\rightarrow 1}$ fails before completing the adjustment of preferred lists and the updating procedure, the tasks in EAQ and BKQ of this node will be lost. Since the communication delay for updating BKQ increases as the number of preferred lists to be adjusted increases, it is important to use an algorithm that requires minimal adjustments.

To further reduce the probability of losing tasks, multiple BKQs can be provided to maintain/update the tasks from a node's second, or higher, preferred nodes. But the communication overhead and delay for maintaining/updating these BKQs in each node may increase to an extent to offset any improvement to be gained by using multiple BKQs. In fact, our simulation results show that the improvement with more than two BKQs in each node is insignificant, as compared to a single BKQ when the number of faulty nodes is less than 25% of the total number of nodes in the system.

5.3.1 Analysis of Average Number of Task Losses

To analyze the average number of task losses with a different number of backup nodes, we need to introduce the following notation:

- $1/\lambda$: mean time between failure (MTBF).
- T_{exe} : (average) task execution time.
- α : ratio of MTBF to T_{exe}.
- β : ratio of task transfer time to T_{exe}.
- A_T : average number of tasks queued in a node.
- P_f : probability of a node failure before completing the updating process.
- $P_f(t_1, t_2)$: probability of a node failure in time interval $[t_1, t_2]$.
- t_u : time to transfer a task between two adjacent nodes.
- $\lambda\, e^{-\lambda\, t}$: probability density function (pdf) of a node failure in [0, t].
- T_{lost} : average number of lost tasks in a node
- $T_q^{N_i}$: N_i's queue for externally arriving tasks.
- $B_{q,j}^{N_i}$: the j^{th} backup queue of N_i.

Single Backup Queue

Suppose N_1 and N_2 back up each other, then we have $B_{q,2}^{N_1} = T_q^{N_2}$ and $B_{q,1}^{N_2} = T_q^{N_1}$. If N_1 becomes faulty, N_2 will treat the tasks in its backup queue as its own externally–arriving tasks, the new task queue of N_2 will be $T_q^{N_2} \cup B_{q,1}^{N_2} = T_q^{N_2} \cup T_q^{N_1}$. Since N_2 was backed up by N_1, it must find a new backup node N_x and copy the new $T_q^{N_2}$ to $B_{q,1}^{N_x}$. If this process is completed before N_2 becomes faulty, no tasks will be lost; otherwise, some of the tasks will be lost. Let the time of N_1's failure be the reference time '0'. Since the average number of tasks in $T_q^{N_2}$ is $2A_T$ after N_1 becomes faulty, $2A_T$ tasks will be lost if N_2 fails between $[0, t_u]$, $2A_T - 1$ tasks will be lost if N_2 fails between $[t_u, 2t_u]$, and so on. The average number of tasks lost is derived as:

$$\begin{aligned}
T_{lost} &= 2A_T\, P_f(0,t_u) + (2A_T - 1)\,(1 - P_f(0,t_u))\, P_f(t_u, 2t_u) + \cdots \\
&\quad +2\,[1 - P_f(0,(2A_T - 2)t_u)]\, P_f((2A_T - 2)t_u, (2A_T - 1)t_u) + \\
&\quad [1 - P_f(0,(2A_T - 1)t_u)]\, P_f((2A_T - 1)t_u, 2A_T t_u) \\
&= 2A_T\, P_f(0,t_u) + \sum_{i=1}^{2A_T-1} (2A_T - i)\,[1 - P_f(0, it_u)]\, P_f(it_u, (i+1)t_u). \qquad (5.1)
\end{aligned}$$

From the definition of $P_f(t_1, t_2)$, we have

$$\begin{aligned}
P_f(t_1, t_2) &= \int_{t_1}^{t_2} \lambda\, e^{-\lambda t} = e^{-\lambda t_1} - e^{-\lambda t_2} \\
&\approx \lambda(t_2 - t_1) \quad \text{when } \lambda\, t_1, \lambda\, t_2 \ll 1.
\end{aligned}$$

Then, we have

$$\begin{aligned}
P_f(0, mt_u) &= m\,\lambda t_u \\
P_f(mt_u, (m+1)t_u) &= \lambda t_u.
\end{aligned}$$

Using the above equation, we can rewrite Eq. (5.1) as follows.

$$\begin{aligned}
T_{lost} &= 2A_T\, \lambda t_u + (2A_T - 1)\,(1 - \lambda t_u)\, \lambda t_u + \cdots \\
&\quad +2\,[1 - (2A_T - 2)\lambda t_u]\, \lambda t_u + [1 - (2A_T - 1)\lambda t_u]\, \lambda t_u \\
&= \lambda t_u \left[\sum_{i=1}^{2A_T} i - \lambda t_u \sum_{i=1}^{2A_T-1} (2A_T - i)\, i \right] \\
&= \lambda t_u \left[A_T\,(2A_T + 1) - \lambda t_u\, \frac{A_T\,(2A_T - 1)(2A_T + 1)}{3} \right] \\
&\approx [A_T\,(2A_T + 1)]\, \lambda t_u \quad \text{when } \lambda t_u \ll 1 \qquad (5.2)
\end{aligned}$$

Double Backup Queues

Let N_1, N_2, and N_3 back up each other, then $B_{q,1}^{N_1} = T_q^{N_2}$, $B_{q,2}^{N_1} = T_q^{N_3}$, $B_{q,1}^{N_2} = T_q^{N_1}$, and $B_{q,2}^{N_3} = T_q^{N_1}$. Note that the second (first) backup queue of N_2 (N_3) will be the task queue of some other node. If N_1 becomes faulty, N_2 will treat the tasks in its backup queue as its own tasks, the new task queue of N_2 will be $T_q^{N_2} \cup B_{q,1}^{N_2} = T_q^{N_2} \cup T_q^{N_1}$. Since N_2's first backup queue was in N_1, it must find a new first backup node and copy its $T_q^{N_2}$ to the newly-selected node. If N_2 failed before completing this process, N_3 will take over its $B_{q,2}^{N_3}$, and the same process will start on N_3. However, if N_3 failed before completing this process, some of tasks in N_1 will be lost. The probability of losing tasks is analyzed as follows. Suppose N_2 fails in $[0, t_u]$, then

$$\begin{aligned} T_{lost}^{(1)} &= P_f(0,t_u) \sum_{j=0}^{A_T-1} (A_T - j)\,[1 - P_f(0,jt_u)]\,P_f(jt_u,(j+1)t_u) \\ &= (\lambda t_u)^2 \sum_{j=0}^{A_T-1} (A_T - j)\,(1 - j\lambda t_u). \end{aligned}$$

Suppose N_2 fails in $[t_u,\ 2t_u]$, then

$$\begin{aligned} T_{lost}^{(2)} &= (1 - P_f(0,t_u))\,P_f(t_u,2t_u) \sum_{j=0}^{A_T-2} (A_T - j - 1)\,[1 - P_f(0,\,jt_u)]\,P_f(jt_u,\,(j+1)t_u) \\ &= (1 - \lambda t_u)\,(\lambda t_u)^2 \sum_{j=0}^{A_T-2} (A_T - j - 1)\,(1 - j\lambda t_u). \end{aligned}$$

Since N_2 can fail at any time in $[0,\ A_T t_u]$, we have

$$\begin{aligned} T_{lost} &= \sum_{i=0}^{A_T-1} (1 - P_f(0,it_u))\,P_f(it_u,(i+1)t_u) \times \qquad (5.3) \\ &\qquad \sum_{j=0}^{A_T-i-1} (A_T - i - j)\,[1 - P_f(0,jt_u)]\,P_f(jt_u,(j+1)t_u) \\ &= (\lambda t_u)^2 \sum_{i=0}^{A_T-1} -\frac{(\lambda t_u)^2}{6}\,i^4 + \frac{\lambda t_u}{6}\,[(3\,A_T - 1)\,\lambda t_u) - 5]\,i^3 \\ &\quad + \left[1 - \frac{\lambda t_u}{3} - \frac{(\lambda t_u)^2}{6} + A_T\,\lambda t_u(\frac{3}{2} + \frac{\lambda t_u}{3} - \frac{A_T}{2}\,\lambda t_u\right]\,i^2 \\ &\quad + \left[\frac{1}{2} - 2\,A_T - \frac{\lambda t_u}{6} - \frac{(\lambda t_u)^2}{6} + (\frac{1}{6} + \frac{A_T}{2})A_T\,\lambda t_u + (1 - \frac{A_T}{6} + \frac{A_T^2}{6})\,\frac{A_T}{6}(\lambda t_u)^2\right]\,i \\ &\quad + A_T^2(1 + \frac{\lambda t_u}{6} - \frac{\lambda t_u}{6}A_T) - \frac{1}{2}(1 + \frac{\lambda t_u}{3})(A_T - 1)) \\ &\approx (\lambda t_u)^2 \left[\sum_{i=0}^{A_T-1} i^2 + (\frac{1}{2} - 2\,A_T)i + A_T^2 - \frac{A_T - 1}{2}\right] \\ &= (\lambda t_u)^2\,A_T\left(\frac{1}{3}A_T^2 + \frac{A_T}{4} + \frac{5}{12}\right). \qquad (5.4) \end{aligned}$$

Eq. (5.3) gives the probability of task loss when N_1's failure is followed by N_2 and N_3. Since N_2 and N_3 both have one backup queue on N_1, if the nodes that hold another backup queue of N_2 or N_3 fail, the primary tasks in N_2 and N_3 will be lost. This probability can be expressed exactly as Eq. (5.3). So, the total probability of task loss is three times of Eq. (5.3). However, there are many higher-order probabilities of task loss. That is, when N_1, N_2, N_3, N_4, and N_5 all failed, if the nodes that hold N_4 and N_5's backup queue fail, the primary tasks in N_4 and N_5 will also be lost, and so on. Since there are many combinations (2^n for an n-cube) that will result in higher-order probabilities of losing tasks and these probabilities decrease exponentially as the number of nodes involved increases, the combinations with more than 5 nodes are ignored in the analysis.

Triple Backup Queues

In the case of three backup queues, the probability of task loss can be expressed as follows. (Due to its complexity, a closed-form solution cannot be found.)

$$\begin{aligned}
T_{lost} &= \sum_{i=0}^{A_T-1} (1 - P_f(0, it_u))\, P_f(it_u, (i+1)t_u) \sum_{j=0}^{A_T-i-1} (A_T - i - j)\,[1 - P_f(0, j\,t_u)] \\
&\quad P_f(jt_u, (j+1)t_u) \sum_{k=0}^{A_T-i-j-1} (A_T - i - j - k)\,[1 - P_f(0, kt_u)]\, P_f(kt_u, (k+1)\,t_u) \\
&= \sum_{i=0}^{A_T-1} (1 - i\lambda t_u)\,\lambda t_u \sum_{j=0}^{A_T-i-1} (A_T - i - j)\,(1 - j\lambda t_u)\,\lambda t_u \times \\
&\quad \sum_{k=0}^{A_T-i-j-1} (A_T - i - j - k)\,(1 - k\lambda t_u)\,\lambda t_u.
\end{aligned}$$

For the case of multiple backup queues, one can express the probability of task loss similarly to the above equation, but it is too difficult to derive a closed-form solution. Note that λt_u in all equations is equal to β/α.

5.3.2 Simulation Results

In addition to modeling, the performance of the proposed adjustment algorithms and fault-tolerant BKQ mechanisms are also evaluated via simulations. The results in [25] show that threshold pattern '1 2 3' performs well in a wide range of system load, and thus, is used in the simulations. The size of buddy set is chosen to be 10, because the performance improvement beyond this size is shown to be insignificant [25]. The system load is varied from 0.5 (medium-loaded) to 0.9 (overloaded) and the number of faulty nodes is changed from 5% to 50% of the total number of nodes in an 8–cube system.

The first simulation is run without adjusting preferred lists. Faulty nodes are randomly generated before the simulation and no new faults are assumed to occur during the simulation. Since the faulty nodes are simply dropped from the preferred lists, the missing probability — the probability of a task missing its deadline — increases very fast with the number of node failures when preferred lists are not adjusted. The missing probabilities in the presence of faulty nodes are normalized to the case without faulty nodes in Fig. 5.2. In the case when 50% of nodes are faulty and the system load is 0.8, the missing probability can be two times as high as the case without faulty nodes.

Another simulation is run to test the goodness of Algorithms 5.2. In order to eliminate other factors that may influence the results, the faulty nodes are randomly generated, and the preferred lists of the nodes with faulty nodes in their buddy set are adjusted before the simulation. No new faults are assumed to occur throughout the simulation. These results are superimposed in Fig. 5.2. Surprisingly, the missing probability obtained for the case with faulty nodes is found to be nearly the same as that for the case without faulty nodes regardless of the percentage of faulty nodes and system load.

The performance of the fault-tolerant BKQ method is measured by the number of lost tasks. The tasks in a node and its most preferred node will be lost when these two nodes fail within the time period, T_{adjust}, required to adjust the preferred lists. The simulation results are tabulated in Table 5.3, where three BKQs (= 1, 2, 3) along with the case of BKQ = 0 are listed at each run under different system loads. The analytical results are plotted in Figs. 5.3 and 5.4. The simple fault-tolerant mechanism with BKQ = 1 is shown to completely eliminate the number of lost tasks when the number of faulty nodes is less than 15% of the total number of nodes and the system load is less than 0.7. Except in an extreme case when the number of faulty nodes reaches 50% of the total number of nodes, this simple approach can reduce the number of lost tasks significantly, as compared to the approach without BKQ.

Using multiple BKQs reduces further the number of lost tasks for the cases of a higher percentage of faulty nodes. However, due to the increased communication overhead and delay with the multiple BKQs, the number of lost tasks cannot be completely eliminated when 50% or higher percentage of the total number of nodes are faulty. As shown in Fig. 5.3 the cases of BKQ=2 and 3 yield more tasks losses as compared to BKQ=1 when the node is not too reliable ($MTBF = 200T_{exe}$). On the other hand, when the node is relatively reliable ($MTBF = 5000T_{exe}$), using of the multiple backup queues can reduce the number of task losses except when $\beta > 0.7$ in the case of the two backup queues.

The analytical results agree well with the simulation results in all cases. For example, consider the case when 35% of nodes are faulty, $\beta = 0.1$, and $\rho = 0.8$. A_T is approximately equal to 1.5 [25] (Table 2.1) and MTBF is close to 5000 T_{exe}. From Fig. 5.4, the average number of tasks lost on a node by using 1, 2, and 3 backup queues are 8×10^{-5}, 1.0×10^{-5}, and 1.1×10^{-7}, respectively. The total number of lost tasks is equal to the above values times the total number of processed tasks which is around 4.1×10^4 in the simulation. So, the total number of lost tasks for 1, 2, and 3 backup queues will be 32.8, 4.1, and 0.045, while the simulation results are 27, 6, and 0, respectively.

An interesting problem found during the simulation is that the probability of missing task deadlines in the case of using BKQs is higher than the case without any

BKQ. The reason is that a node may fail during the processing of a task, and this task is restarted on another node instead of continuing its execution from the point in time when the node failed. Although this task can be successfully completed by another node, the total processing time may often exceed its deadline if it is queued at the new node before its execution.

Preference / # of faulty nodes	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	7	8
2	2	4	6	8	10	12	14	12	12	14
3	3	6	9	12	15	18	21	16	16	20
12	10	19	25	29	36	40	43	60	45	46
38	30	42	47	49	58	55	60	62	58	59
64	36	46	52	51	51	53	52	51	54	52

Table 5.1: Number of nodes selected by more than two nodes as their k^{th} preferred nodes in an 8-cube system with buddy set of 10 nodes where $1 \leq k \leq 10$

	Average distance		Distance to most preferred node		Number of adjustments on a node
	Lower	Upper	Lower	Upper	
Algorithm 1	1.33	2.89	1	4	2
Algorithm 2	1.33	1.46	1	4	2
Algorithm 3	1.33	1.51	1	2	4

Table 5.2: Comparison of perferred list adjustment algorithms in an 8-cube system with 15 faulty nodes

% of faulty nodes						
System load	# of BKQs	5 %	15 %	25 %	35 %	50 %
0.5	0	10	29	36	50	67
	1	0	1	3	8	29
	2	0	0	0	2	10
	3	0	0	0	0	2
0.6	0	14	35	46	74	106
	1	0	3	4	10	35
	2	0	0	0	5	13
	3	0	0	0	0	3
0.7	0	23	46	73	105	180
	1	0	3	4	15	47
	2	0	0	0	5	15
	3	0	0	0	0	5
0.8	0	31	59	87	134	211
	1	0	3	6	27	61
	2	0	0	1	6	26
	3	0	0	0	0	9
0.9	0	42	86	96	164	297
	1	0	5	11	35	84
	2	0	0	5	7	30
	3	0	0	0	1	12

Table 5.3: The number of lost tasks vs. number of backup queues.

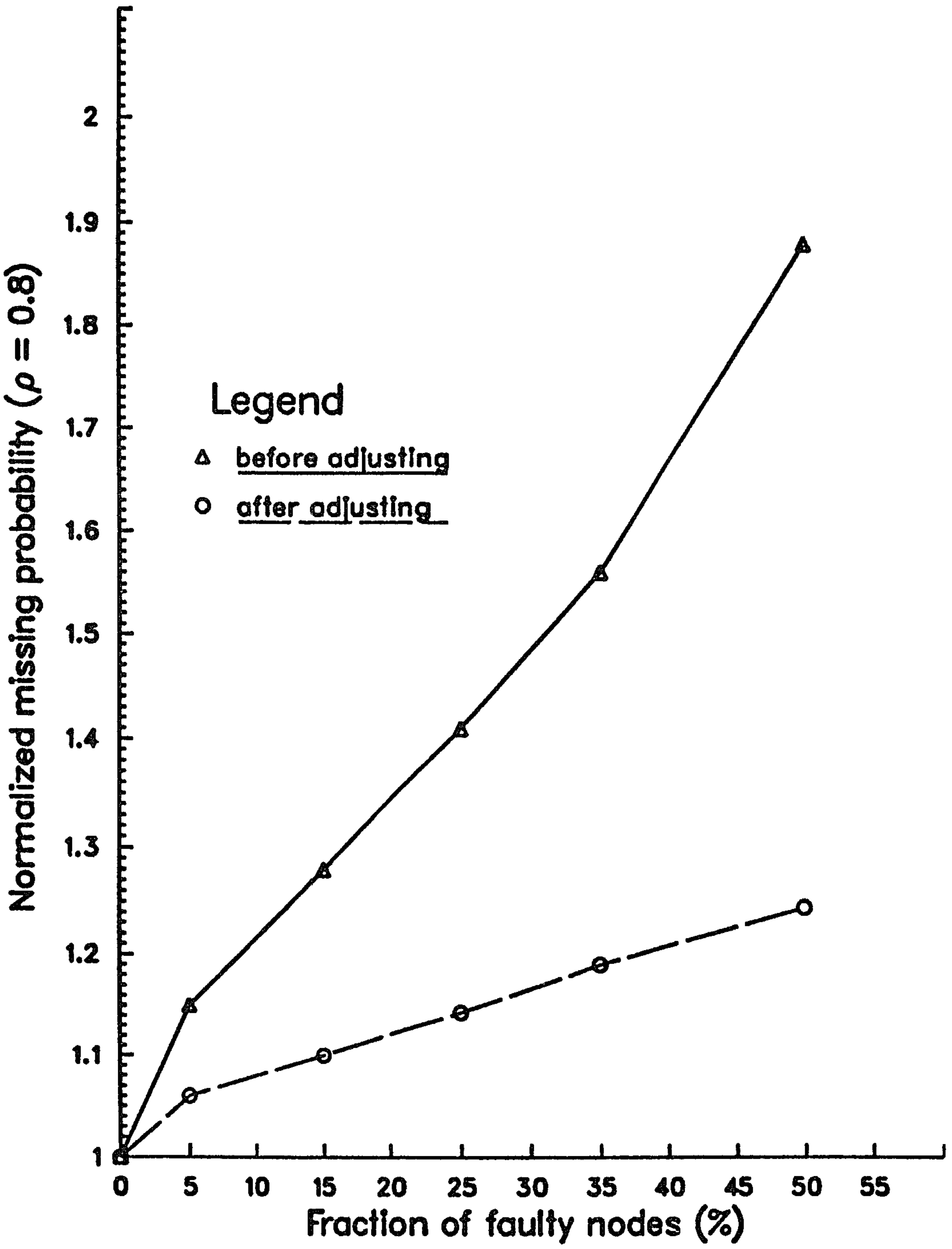


Figure 5.2: Comparison of missing probabilities.

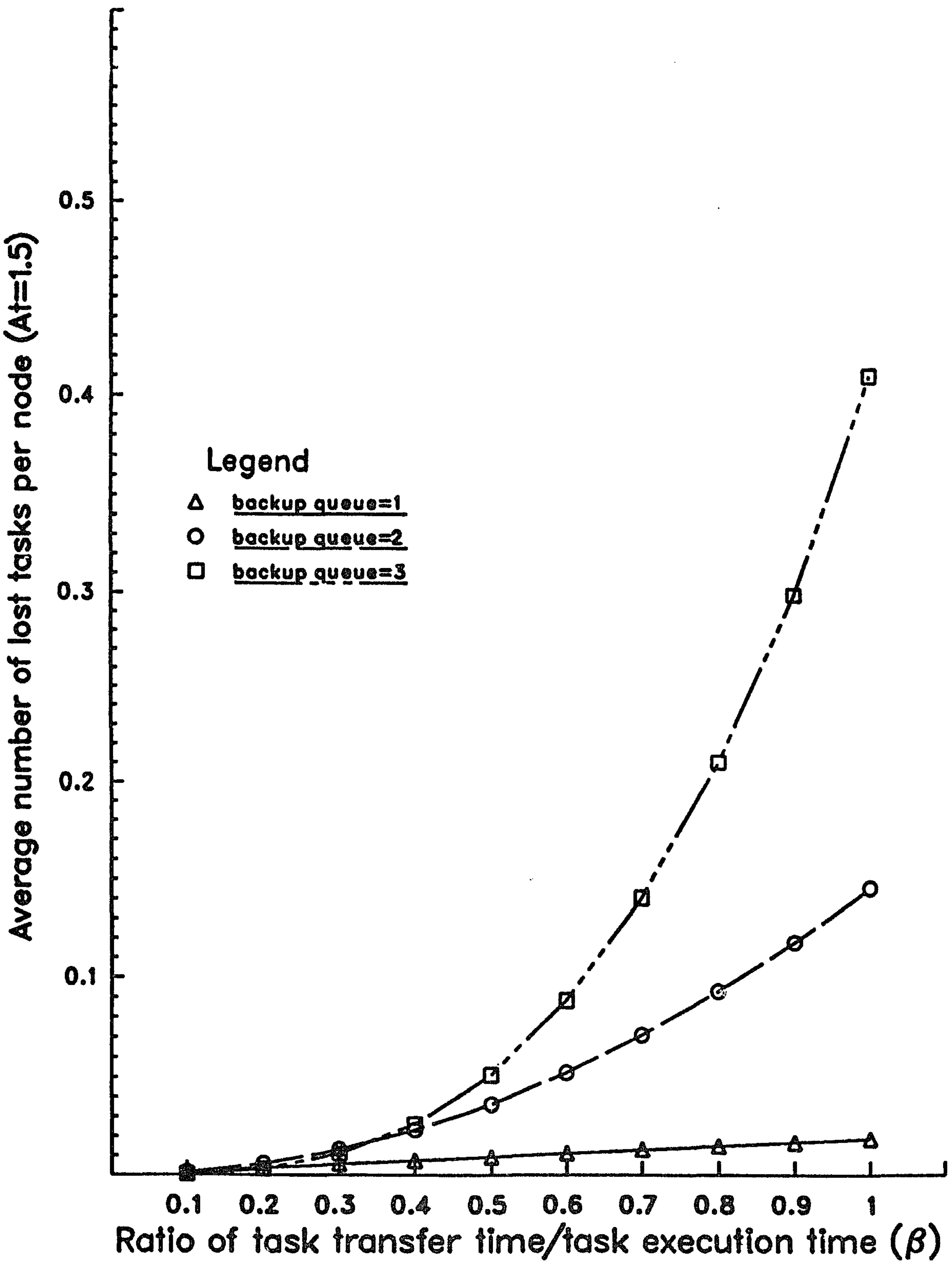


Figure 5.3: Number of lost tasks vs. β ($MTBF = 200 \times T_{exe}$)

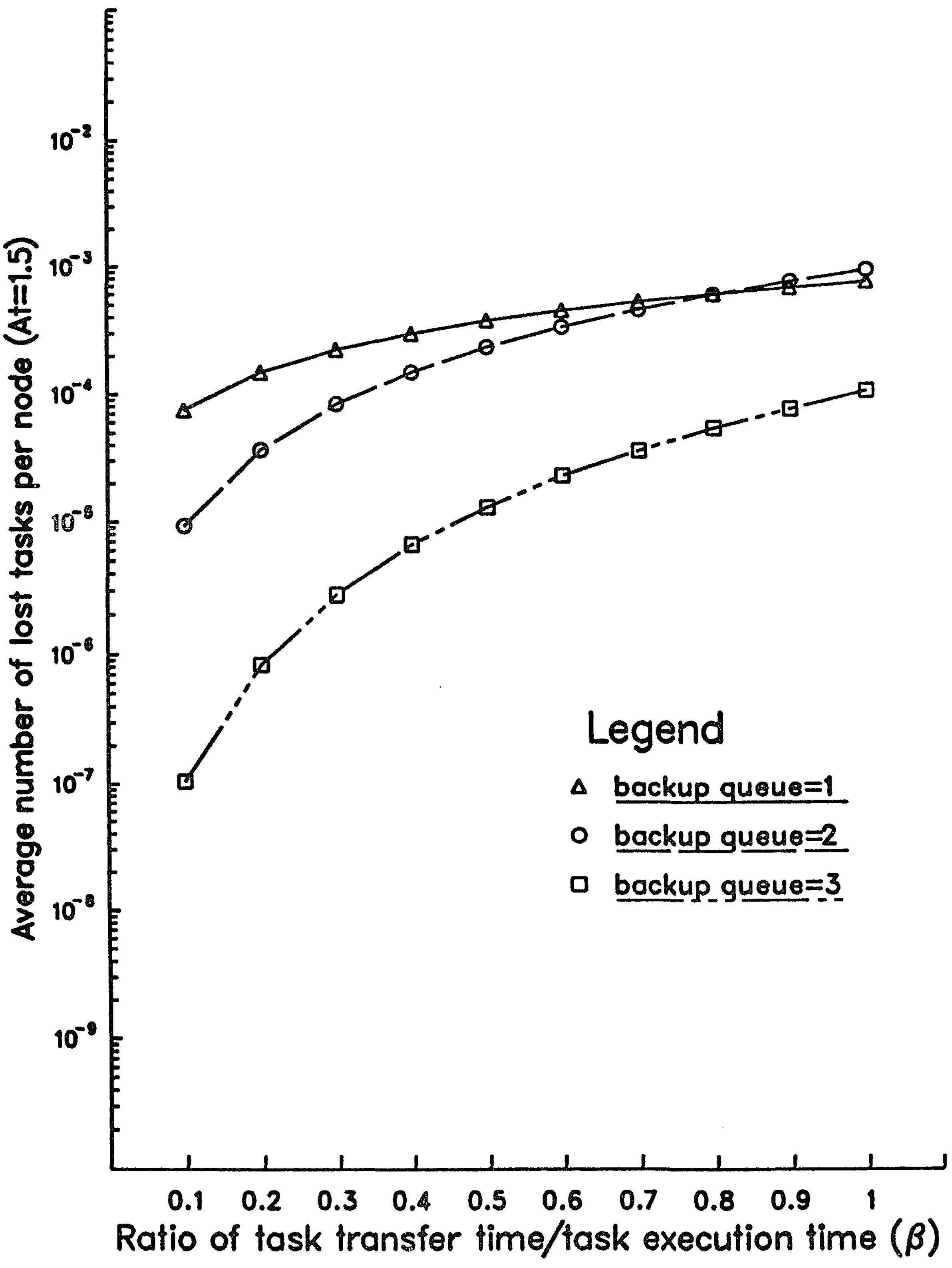


Figure 5.4: Number of lost tasks vs. β ($MTBF = 5000 \times T_{exe}$)

Node	Preferred list in buddy sets with 10 nodes									
0	1	2	4	8	16	32	64	128	6	10
1	0	3	5	9	17	33	65	129	7	11
	19	3	5	9	17	33	65	129	7	11
18	19	16	22	26	2	50	82	146	20	24
	1	16	22	26	2	50	82	146	20	24
2	3	0	6	10	18	34	66	130	4	8
	3	14	6	10	18	34	66	130	4	8
12	13	14	8	4	28	44	76	140	10	6
	13	2	8	4	28	44	76	140	10	6
4	5	6	0	12	20	36	68	132	2	14
	5	6	38	12	20	36	68	132	2	14
34	35	32	38	42	50	2	98	162	36	40
	35	32	4	42	50	2	98	162	36	40
8	9	10	12	0	24	40	72	136	14	2
	9	10	12	74	24	40	72	136	14	2
66	67	64	70	74	82	98	2	194	68	72
	67	64	70	8	82	98	2	194	68	72
16	17	18	20	24	0	48	80	144	22	26
	17	18	20	24	146	48	80	144	22	26
130	131	128	134	138	146	162	194	2	132	136
	131	128	134	138	16	162	194	2	132	136

Figure 5.5: Adjustment of the preferred lists in a 8-cube system with faulty node '0'

Node	Preferred list in buddy sets with 10 nodes									
0	1	2	4	8	16	32	64	128	6	10
32	33	34	36	40	48	0	96	160	38	42
	33	34	36	40	48	52	96	160	38	42
20	21	22	16	28	4	52	84	148	18	30
	21	22	16	28	4	32	84	148	18	30
64	65	66	68	72	80	96	0	192	70	74
	65	66	68	72	80	96	100	192	70	74
36	37	38	32	44	52	4	100	164	34	46
	37	38	32	44	52	4	64	164	34	46
128	129	130	132	136	144	160	192	0	134	138
	129	130	132	136	144	160	192	196	134	138
68	69	70	64	76	84	100	4	196	66	78
	69	70	64	76	84	100	4	128	6	10
6	7	4	2	14	22	38	70	134	0	12
	7	4	2	14	22	38	70	134	142	12
136	137	138	140	128	152	168	200	8	142	130
	137	138	140	128	152	168	200	8	6	10
10	11	8	14	2	26	42	74	138	12	0
	11	8	14	2	26	42	74	138	12	58
48	49	50	52	56	32	16	112	176	54	58
	49	50	52	56	32	16	112	176	54	10

Figure 5.6: Adjustment of the preferred lists in a 8-cube system with faulty node '0' (Continued)

CHAPTER 6

SCHEDULING PERIODIC TASKS WITH CONSIDERATION OF LOAD SHARING OF APERIODIC TASKS

6.1 Introduction

Scheduling both periodic and aperiodic tasks in real–time systems is a much more difficult problem than considering periodic or aperiodic tasks alone [28, 29, 30, 31]. Sprunt *et al.* proposed a scheme that can guarantee the deadlines of periodic tasks while minimizing the response time of aperiodic tasks by exploiting the CPU cycles left unused by periodic tasks [30]. In their study, aperiodic tasks are assumed to have soft deadlines with the lowest priority. One difficulty in scheduling aperiodic tasks in the presence of periodic tasks is the lack of *a priori* knowledge of their interarrival times and deadlines. Although aperiodic tasks can be guaranteed by using the worst–case interarrival time between two aperiodic tasks as the period for all aperiodic tasks and treating them as periodic tasks, the CPU cycles will be greatly wasted in most cases, because the worst–case interarrival time is usually much smaller than the average case. The problem of under–utilizing CPU cycles can be eased by requiring a minimum separation time p for any two aperiodic tasks. Mok showed that aperiodic tasks can be guaranteed if p is large enough [28], but did not discuss the case when this condition does not hold.

The main purpose of this chapter is to propose a new scheduling algorithm, called the *reservation–based* (RB) scheduling algorithm that can guarantee the deadlines of periodic tasks while minimizing the probability of missing the deadlines of aperiodic tasks. In this algorithm, periodic tasks are scheduled according to the rate monotonic priority algorithm (RMPA) [12]. Aperiodic tasks are assumed to have the lowest priority and scheduled by utilizing the CPU time left unused by periodic tasks in each *unit cycle.* A unit cycle u is the greatest common divisor (GCD) of all periods [38]. The CPU utilization, $U(i)$, in unit cycle i is calculated by dividing the used CPU time in that unit cycle by u. In the RB algorithm, a certain fraction of CPU time, denoted by R, is reserved for aperiodic tasks in each unit cycle without violating the deadlines of periodic tasks. So, at least a fraction R of unit cycle will be left unused in each unit cycle after scheduling all periodic tasks. The value of R is found to greatly affect the probability of guaranteeing aperiodic tasks even when the average CPU utilization is fixed. For example, let the period and computation time of task τ_1 be 3 and 1.5 unit cycles, respectively. This task can be scheduled by either allocating 1.0 and 0.5 in the first and second unit cycles, or 0.5 in each unit cycle. However, if an aperiodic task ν_1 with computation time of 1.0 unit cycle and a deadline of 2 unit cycles arrives when τ_1 is initiated, then the first scheduling scheme can only allocate 0.5 unit cycle for ν_1, thus missing its deadline. By contrast, the second scheduling scheme will be able to allocate 1.0 unit cycle to ν_1, thus guaranteeing its deadline. Thus, one of the most important issues in the RB algorithm is to derive the relation between R and the probability of guaranteeing aperiodic tasks. Since the probability of guaranteeing aperiodic tasks is a monotonic increasing function of R, we want to derive the maximum value of R, denoted as R_{max}, which is the maximum fraction of CPU time that can be reserved in each unit cycle without violating the deadline of any periodic task in a given task set.

Another goal in this chapter is to implement the RB scheduling algorithm by modifying the LSMSCB. A set of periodic tasks on each node is pre-assigned and scheduled, and the CPU utilization on each node is known at the time an aperiodic task arrives. R_{max} is derived for a given task set[1]. Upon arrival of an aperiodic task the node will check the CPU utilization at that unit cycle; if the task can be guaranteed locally, the node will schedule this task by using the unused CPU time and broadcast the change of CPU utilization (due to the addition of the task) to the nodes in its buddy set. If this task cannot be guaranteed locally, the node will check the available CPU time on other nodes using its preferred list. If a receiver node is found, this task will be transferred to the selected node, otherwise it is rejected.

The RB algorithm is attractive in scheduling both periodic and aperiodic tasks, because it increases the total number of tasks that can be guaranteed for the following reason. Since periodic tasks are invoked at the beginning of their respective periods, there may be more tasks invoked in some time intervals than others if the first release times of periodic tasks are different. For example, as shown in Fig. 6.1, $U(i)$ varies from 0 to 1.0. Moreover, the CPU utilization by periodic tasks is 100% from unit cycle 15 to 18, while it is only 50% from unit cycle 25 to 28. The variation of $U(i)$ is not desirable in any real–time system, because the probability of guaranteeing an aperiodic task depends not only on its deadline but also on its arrival time. Since the variation of $U(i)$ results from the RMPA algorithm, some other schemes must be used to reduce the variation of $U(i)$. The earliest deadline (ED) algorithm is not suitable for scheduling both periodic and aperiodic tasks for the following reason. Since under the RMPA algorithm, the lower priority task cannot preempt the higher priority task, and since aperiodic tasks have the lowest priority and cannot preempt any of the periodic tasks, their deadlines cannot be guaranteed by the ED algorithm. By contrast, the RB algorithm reduces the variation of $U(i)$ in each unit cycle, thus increasing the total number of tasks that can be guaranteed.

Section 6.2 states the problem and reviews some related results in [12]. Section 6.3 presents the RB scheduling algorithm and an analytic model for its performance analysis.

[1]each node may have a different task set

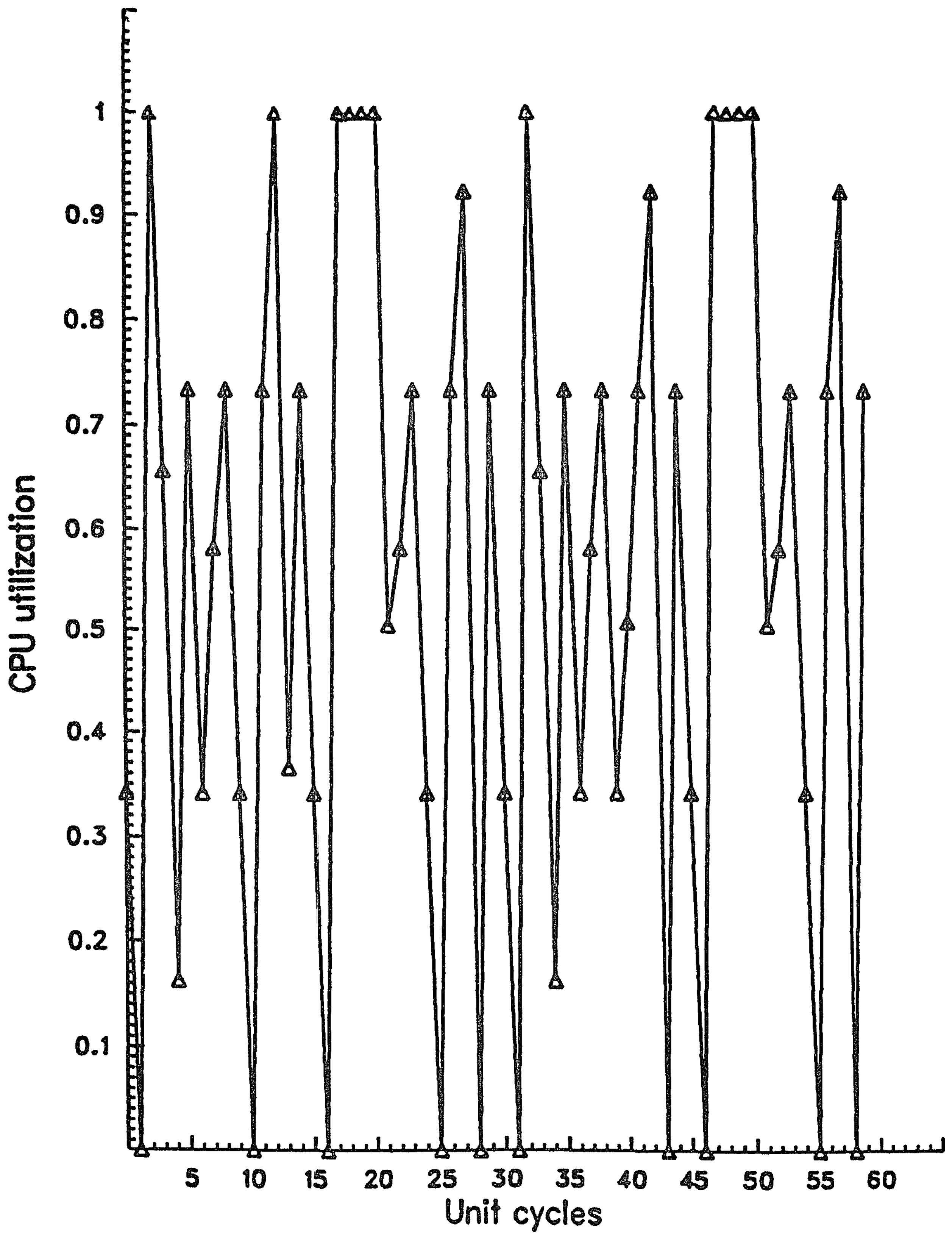


Figure 6.1: CPU utilization with $U = 0.6$ and $R = 0.0$.

Section 6.4 discusses the implementation of the proposed algorithm in conjunction with our load sharing method discussed in Chapter 2. The performance of the RB scheduling algorithm is evaluated via both simulation and analytic modeling.

6.2 Reserved Fraction of CPU Time

We want to guarantee the deadlines of periodic tasks by using the RMPA algorithm while maximizing the probability of guaranteeing aperiodic tasks. Some of the basic definitions in [12] are introduced first before presenting the RB scheduling algorithm.

Let aperiodic task ν_i be represented by a three tuple (a_i, C_i, D_i), where a_i is its release time, C_i is the computation time, and D_i is the deadline relative to a_i measured in unit cycles. For periodic task τ_i, the release time is at the beginning of its period and the deadline of each invocation of τ_i is assumed to be the beginning of the next period. τ_i is represented by a three tuple (r_i, C_i, T_i), where r_i is the first release time of τ_i, C_i is the computation time, and T_i is the period. So, the i^{th} invocation of τ_i is released at $r_i + (i-1)T_i$. All periodic tasks are given *a priori* and the priority of a periodic task is determined by its period; the shorter the period the higher its priority. Every periodic task is assumed to arrive at the beginning of a unit cycle [38], but the computation time can be a fraction of one unit cycle. All periodic and aperiodic tasks may be preempted at any time. Since real-time tasks must usually be stored in main memory before putting the system in operation, the time to switch between tasks is assumed negligible.

Using the same notation in [12], the CPU utilization U in executing a set of m tasks is

$$U = \sum_{i=1}^{m} \frac{C_i}{T_i}. \tag{6.1}$$

The *major cycle*, T_{mc}, for a set of tasks is the least common multiple of all task periods in the set measured in number of unit cycles [38]. For example, the i^{th} major cycle starts at $t = (i-1)T_{mc}$ and ends at $t = i\,T_{mc}$. Eq. (6.1) gives the average CPU utilization in a node over one major cycle for a given set of periodic tasks. Let s_i denote the i^{th} unit cycle ($(i-1)\,u \le t < i\,u$) in a major cycle. The CPU utilization in s_i, denoted by $U(i)$, is calculated by dividing the processor busy time in s_i by u. So, the unused (used) CPU time in s_i is $u\,(1-U(i))$ $(u\,U(i))$. From Eq. (6.1) and the definition of $U(i)$,

$$U = \sum_{i=1}^{m} \frac{C_i}{T_i} = \frac{\sum_{i=1}^{T_{mc}/u} U(i)}{T_{mc}}.$$

It is not sufficient to analyze the probability of guaranteeing aperiodic tasks by using U alone. For example, consider the average CPU utilization over N unit cycles for different time intervals in Fig. 6.2 while changing N from 2 to 20. Although $U = 0.6$, the average CPU utilization over a small N, such as $N < 6$, can be as high as 100% or as low as 1%. Thus, the probability of guaranteeing the aperiodic tasks with deadlines shorter than 6 unit cycles will greatly depend on their arrival times. This is not desirable in any real–time system, because the probability of guaranteeing an aperiodic task depends not only on its deadline but also on its arrival time. The difference between the maximum and minimum average utilization over N unit cycles for different time intervals reduces to within 30% of each other only when $N > 13$.

As mentioned earlier, the fraction of CPU time reserved in each unit cycle will greatly affect the probability of guaranteeing aperiodic tasks even when U is fixed. The importance of the reserved fraction of CPU time to the guarantee of aperiodic tasks can be seen from the example in Fig. 6.3 (assuming $u = 1.0$ for convenience). Let $\tau_1 = (0, 1.5, 3)$ and $\tau_2 = (0, 0.5, 5)$. As shown in Fig. 6.3, for an aperiodic task $\nu_3 = (0, 0.6, 2)$ only the third scheduling scheme can allocate 0.6 to complete ν_3 before its deadline, although all three scheduling schemes guarantee the deadlines of τ_1 and τ_2. However, if the deadline of ν_3 is 3 instead of 2, it will be guaranteed by all three scheduling schemes. In this case, even if the computation time of ν_3 is increased to 1.0, it will still be guaranteed by all three scheduling schemes. We define the reserved fraction, R, of the CPU time as the minimal fraction of CPU time left unused in each unit cycle after scheduling all periodic tasks; that

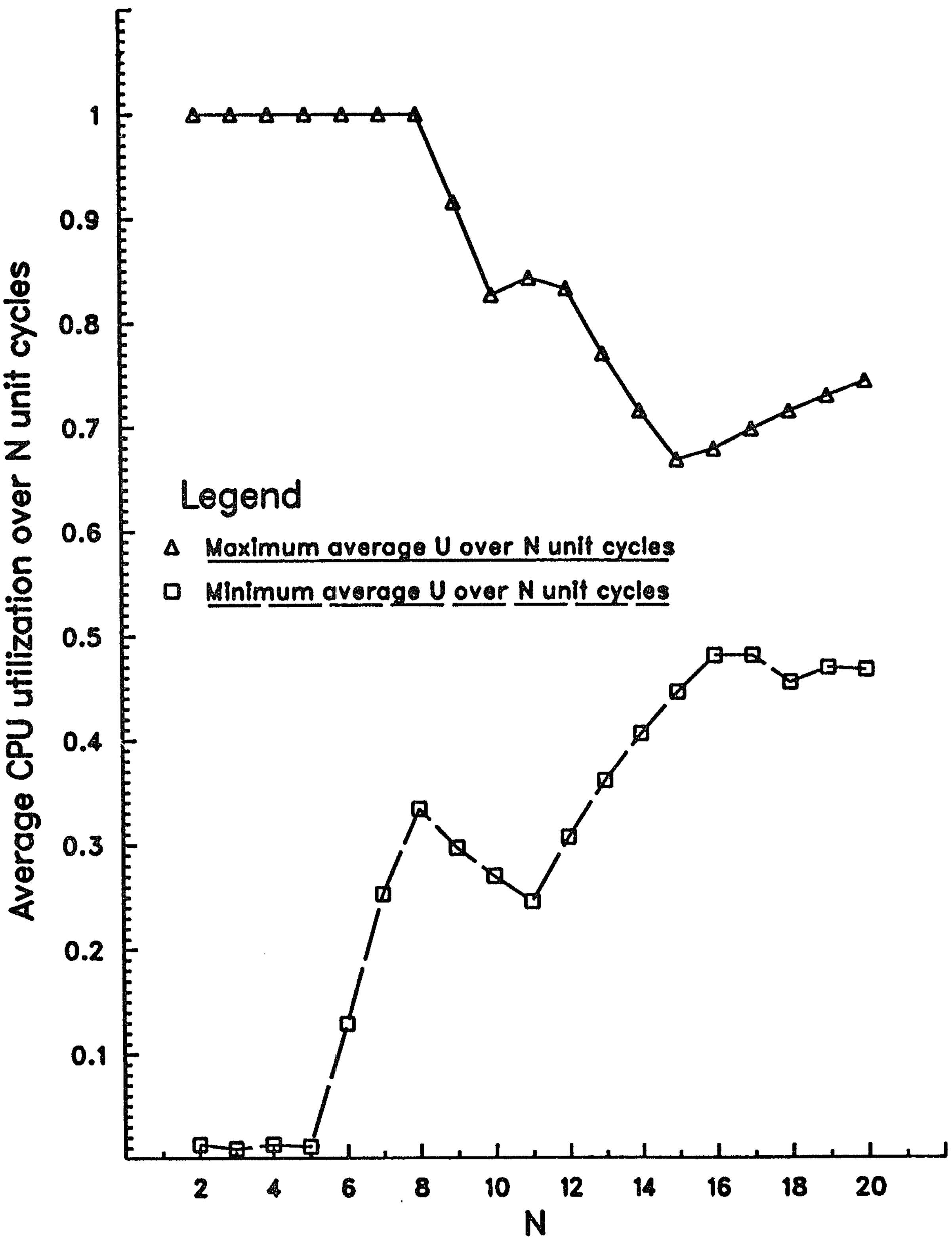


Figure 6.2: Variation of average CPU utilization over N unit cycles ($U = 0.6, R = 0.0$).

is, $R = 1 - \max_i U(i)$. For the example in Fig. 6.3(a), no time is reserved in s_1 and s_2, while 0.5 unit cycle is reserved in s_2 in (b), and 0.3 unit cycle is reserved in each unit cycle in (c). Note that reserving a fraction, R, of each unit cycle does not always guarantee the aperiodic tasks. For example, the first and second scheduling schemes can guarantee an aperiodic task $\nu_4 = (1, 1.0, 2)$, but not the third scheme even though $R = 0.3$. There is also an upper bound of R in order to guarantee the deadlines of all periodic tasks. Consider the third scheduling scheme in Fig. 6.1(c); if $R = 0.4$ from s_1 to s_3, then only 40% of unit cycle can be allocated to τ_2 before its next invocation, thus missing the deadline.

If U is fixed, the value of R will affect the probability of guaranteeing aperiodic tasks with short deadlines and may not affect the tasks with long deadlines. For example, consider the average CPU utilization over $2 \leq N \leq 20$ unit cycles with $R = 0.3$ in Fig. 6.4. The maximum and minimum $U(i)$ are 70% and 1%, respectively, for $N < 6$, while they were 100% and 1%, respectively for the example in Fig. 6.2. However, the difference between the maximum and minimal $U(i)$ is about the same in Figs. 6.2 and 6.4 for $N > 12$. Since reserving a fraction, R, of each unit cycle does not change U, the value of R has less effect on the amount of CPU time that can be allocated for aperiodic tasks with long deadlines.

It should be noted that in the proposed scheme, periodic tasks are scheduled according to the RMPA algorithm. If no aperiodic task arrives, there is no need to reserve any fraction of CPU time in each unit cycle. However, as soon as an aperiodic task arrives at a node, it will check its unused CPU time. If the task can be guaranteed by the node, it will be scheduled locally. Upon completion of an aperiodic task, periodic tasks will be scheduled back according to the RMPA algorithm if there is no aperiodic task in the queue; otherwise the RB algorithm will be used to process the remaining aperiodic tasks.

In the RB scheduling algorithm, $U(i) \leq 1 - R$ for all i after scheduling the periodic tasks according to the RMPA algorithm. To analyze the performance of the RB scheduling algorithm, we must know the pattern of deadlines of aperiodic tasks, which can

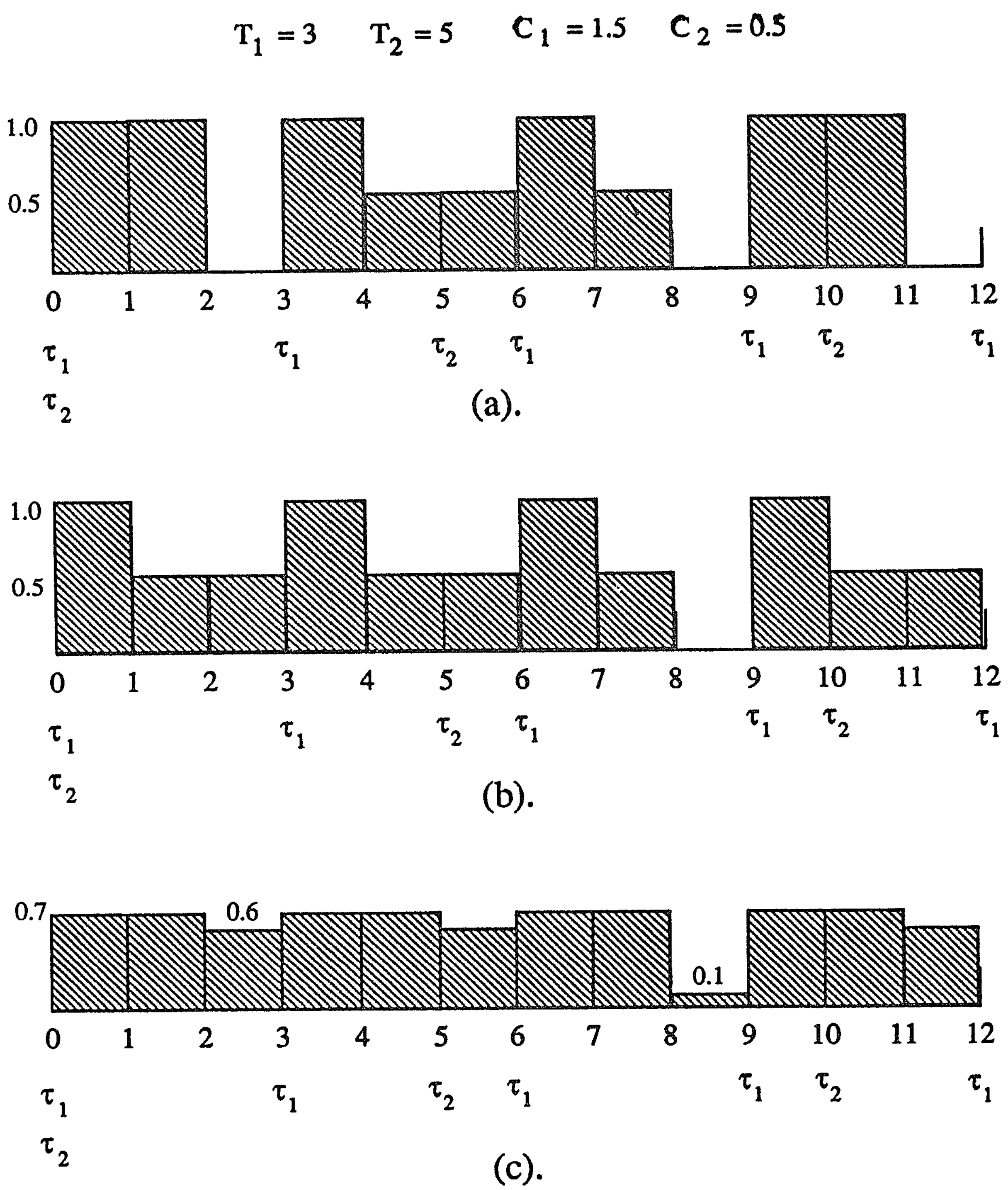


Figure 6.3: Three scheduling schemes with different CPU reservation ratios.

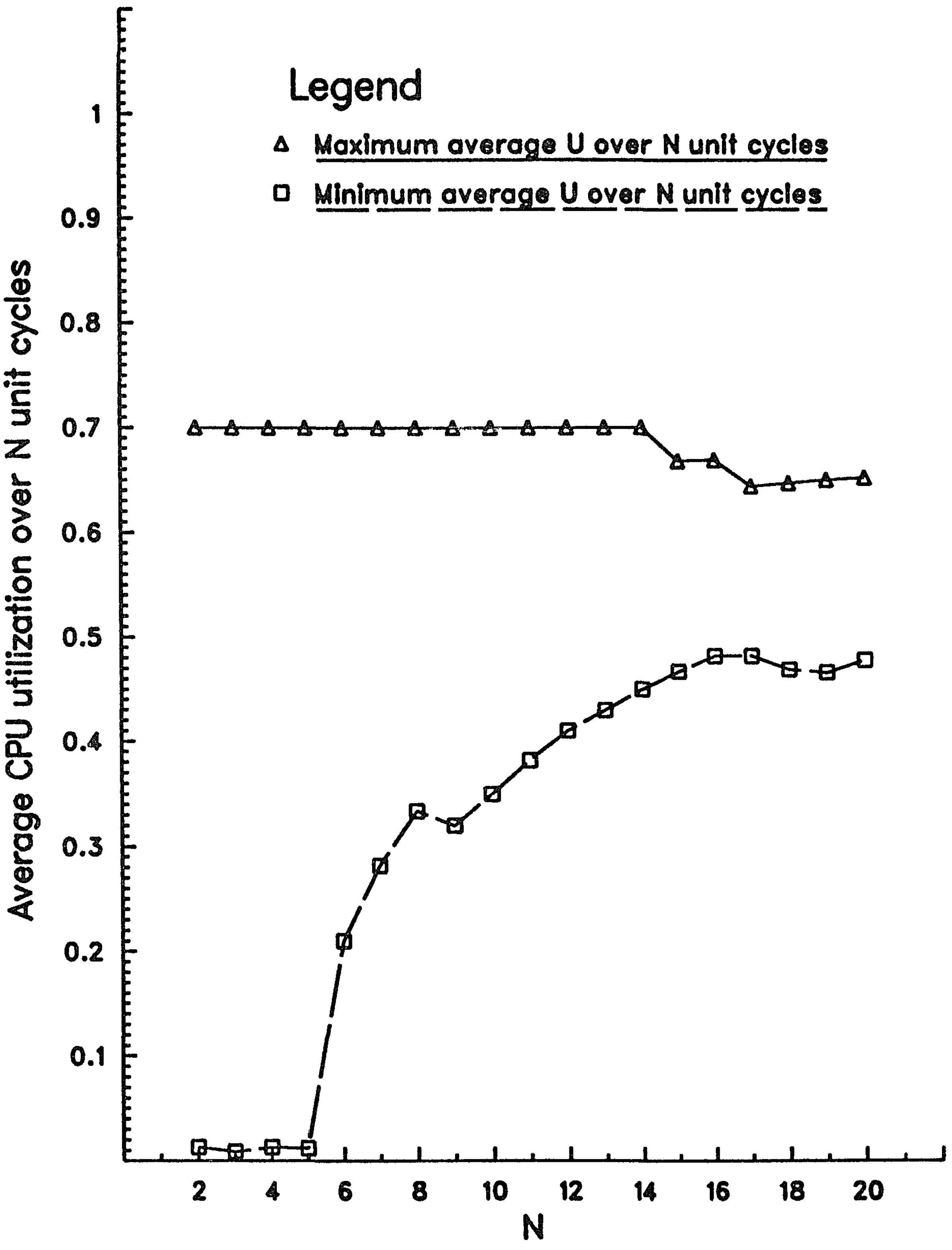


Figure 6.4: Variation of average CPU utilization over N unit cycles ($U = 0.6, R = 0.3$).

be specified by a probability distribution. The probability of guaranteeing an aperiodic task $\nu_i = (a_i, C_i, D_i)$ is the probability of allocating an amount of CPU time $\geq C_i$ after a_i but before or on $a_i + D_i - C_i$. Thus, the deadline distribution of aperiodic tasks is one of the main factors that determine the performance of the RB scheduling algorithm. To implement the RB scheduling algorithm, one must consider:

- Derivation of the CPU utilization in each unit cycle after scheduling a given set of periodic tasks, and the relation between R and the probability of guaranteeing aperiodic tasks.
- Determination of the value of R that maximizes the probability of guaranteeing aperiodic tasks while guaranteeing all periodic tasks.
- The deadline distribution of aperiodic tasks for performance analysis.

6.3 The RB Algorithm and Analytic Modeling of Its Performance

We will first introduce the RB scheduling algorithm and then derive the CPU utilization in each unit cycle.

Each periodic task $\tau_i = (r_i, C_i, T_i)$ is assumed to arrive at the beginning of a unit cycle, but C_i can be a fraction of one unit cycle. So, the first release time r_i of τ_i is a random number between 0 and $T_{mc} - 1$ if the first release times of periodic tasks are different; $r_i = 0$ if all periodic tasks are released at the beginning of each major cycle. A set, S_p, of m periodic tasks are sorted in ascending order of their periods, where $T_1 < T_2 < \cdots < T_m$. Since the first release time r_i ranges from 0 to $T_{mc} - 1$, $U(i)$ will repeat itself after the second major cycle, i.e., $U(i + T_{mc}) = U(i + n\,T_{mc})$ for all $n > 1$. Note that $U(i)$ in the first major cycle may not be equal to $U(i + T_{mc})$ for some i, because the periodic tasks with $r_i > i$ are not yet released in the first major cycle. Thus, it is sufficient to consider periodic tasks for 2 major cycles to derive $U(i)$ after all periodic tasks are released. In the RB algorithm, the initial value of $U(i)$ is set to R for $i = 0, 1, \ldots, 2T_{mc}$. After scheduling all periodic tasks, R

is subtracted from $U(i)$. Since at most $1 - R$ fraction of CPU time in each unit cycle can be allocated for periodic tasks, R is the minimal unused fraction of each unit cycle. If the current invocation of a task cannot be completed before the next invocation, the task set is unschedulable and the RB scheduling algorithm terminates.

RB scheduling Algorithm

```
for $i := 0$ to $2T_{mc}$ do
    $U(i) := R$;
end_do
for $i := 1$ to $|S_p|$ do
    for $s_i := r_i$ to $2T_{mc}$ do
        schedule_time := $C_i$;
        $s' := s_i$;
        while schedule_time $> 0$ do
            $F(s') := (1 - U(s'))u$;
            if schedule_time $\leq F(s')$ then
                $U(s') := U(s')$ + schedule_time/$u$;
            else
                if $F(s') > 0$ then
                    schedule_time := schedule_time $- F(s')$
                    $U(s') := U(s') + F(s')/u$
                end_if
                $s' \longleftarrow s' + 1$;
                if $s' \geq s_i + T_i$, stop, task set unschedulable;
            end_if
        end_do
    end_do
end_do
for $i := 0$ to $2T_{mc}$ do
    $U(i) := U(i) - R$;
end_do
```

The RB scheduling algorithm schedules a set of periodic tasks according to the RMPA algorithm while reserving a fraction R of each unit cycle. In the RMPA algorithm, the task with the shortest period is scheduled first, so the assignment starts from τ_1 in the RB scheduling algorithm. The scheduling of τ_1 starts from its first release time r_1. The scheduled time for τ_1 ("schedule_time") is set to C_1 first, then the unused CPU time at r_1 ($F(s')$) is compared with the τ_1's scheduled time. If $F(s') \geq$ schedule_time, the schedule_time will be added to $uU(s')$ and the scheduling of τ_1 is done; otherwise, $F(s')$ is subtracted from schedule_time and s' is incremented by one. The above procedure continues until schedule_time becomes 0.

6.3.1 The CPU Utilization for a Set of Periodic Tasks

In order to derive the relation between R and the probability of guaranteeing aperiodic tasks, it is necessary to derive $U(i)$. From [12], if $U \leq m(2^{1/m} - 1)$ for a set S_p of m periodic tasks, the task set is schedulable. Let $\tau_1, \ldots, \tau_m$ be the tasks in S_p. These tasks are sorted in ascending order of their periods, i.e., $T_1 < T_2 < \cdots < T_m$. After τ_1 is scheduled, $U(i)$ can be calculated as follows.

If $C_1 < u(1-R)$, then

$$U(r_i + nT_1) = C_1/u, \qquad n = 0, 1, 2, \ldots$$

If $C_1 > u(1-R)$, then

$$\begin{aligned} U(i + nT_1) &= 1 - R, \qquad i = r_1, r_1 + 1, \ldots, r_1 + \lfloor \frac{C_1}{u(1-R)} \rfloor - 1 \\ U\left(r_1 + \lfloor \frac{C_1}{1-R} \rfloor + nT_1\right) &= \frac{C_1 - \lfloor \frac{C_1}{1-R} \rfloor \times (1-R)}{u}, \qquad n = 0, 1, 2, \ldots \end{aligned}$$

Similarly, $U(i)$ can be calculated after scheduling τ_j, $j = 2, \ldots, k$ as follows. Let ℓ_n be the minimum number of unit cycles needed to execute τ_j at its n^{th} invocation and A_j be the sum of the unused CPU times within this number of unit cycles and $F(i)$ be the unused CPU time in unit cycle i. Then we have $A_j \leq \sum_{i=r_j+nT_j}^{\ell_n} F(i)$ and $A'_j = \sum_{i=r_j+nT_j}^{\ell_n - 1} F(i)$, $n = 0, 1, \ldots, \infty$.

If $C_j/u < 1 - U(r_j + nT_j)$, $n = 0, 1, \ldots$, then

$$U(r_i + nT_j) = U(i + nT_j) + C_j/u, \qquad n = 0, 1, \ldots$$

If $C_j/u > 1 - U(r_j + nT_j)$, $n = 0, 1, \ldots$, then

$$\begin{aligned} U(i + nT_j) &= 1 - R, \qquad i = r_j, \ldots, r_j + \ell_n - 1 \\ U(r_j + \ell_n + nT_j) &= U(r_j + \ell_n + nT_j) + \frac{A_j - A'_j}{u}, \qquad n = 0, 1, \ldots \end{aligned} \tag{6.2}$$

The calculated values of $U(i)$ for a task set with $U = 0.6$ and $R = 0.1, 0.2, 0.3$ are plotted in Figs. 6.5–6.7. From these figures, it is found that many unit cycles are under-utilized or even unused even though the average CPU utilization (U) is around 60%. This observation justifies the approach of reserving a certain fraction of each unit cycle to improve the overall CPU utilization. Another interesting result found in the RB scheme is that the variation of $U(i)$ decreases as R increases, thus making $U(i)$ closer to U. As a result, the probability of guaranteeing aperiodic tasks will be less dependent on their arrival time, because a minimal fraction R of CPU time is reserved in each unit cycle for aperiodic tasks (Figs. 6.5–6.7). Recall that when no CPU time was reserved, the $U(i)$ can be as high as 100% in some unit cycles; thus the probability of guaranteeing aperiodic tasks depends not only on their deadline but also on their arrival time as shown in Figs. 6.1 and 6.2.

6.3.2 Probability of Guaranteeing an Aperiodic Task

The probability of guaranteeing an aperiodic task, $\nu_\alpha = (a_\alpha, C_\alpha, D_\alpha)$, is the probability that C_α is less than or equal to the sum of the unused CPU times during the interval $[a_\alpha,\ a_\alpha + D_\alpha]$. For convenience, C_α is assumed to have an exponential distribution with mean μ. Then,

$$\begin{aligned} P(C_\alpha \leq \sum_{i=a_\alpha}^{a_\alpha+D_\alpha} F(i)) &= P(C_\alpha \leq \overline{F}\, D_\alpha) \\ &= \sum_{D=D_{min}}^{D_{max}} P(D) \times \int_0^{\overline{F}\, D} \mu\, e^{-\mu t} dt \\ &= \sum_{D=D_{min}}^{D_{max}} P(D)(1 - e^{-\mu \overline{F} D}), \end{aligned} \tag{6.3}$$

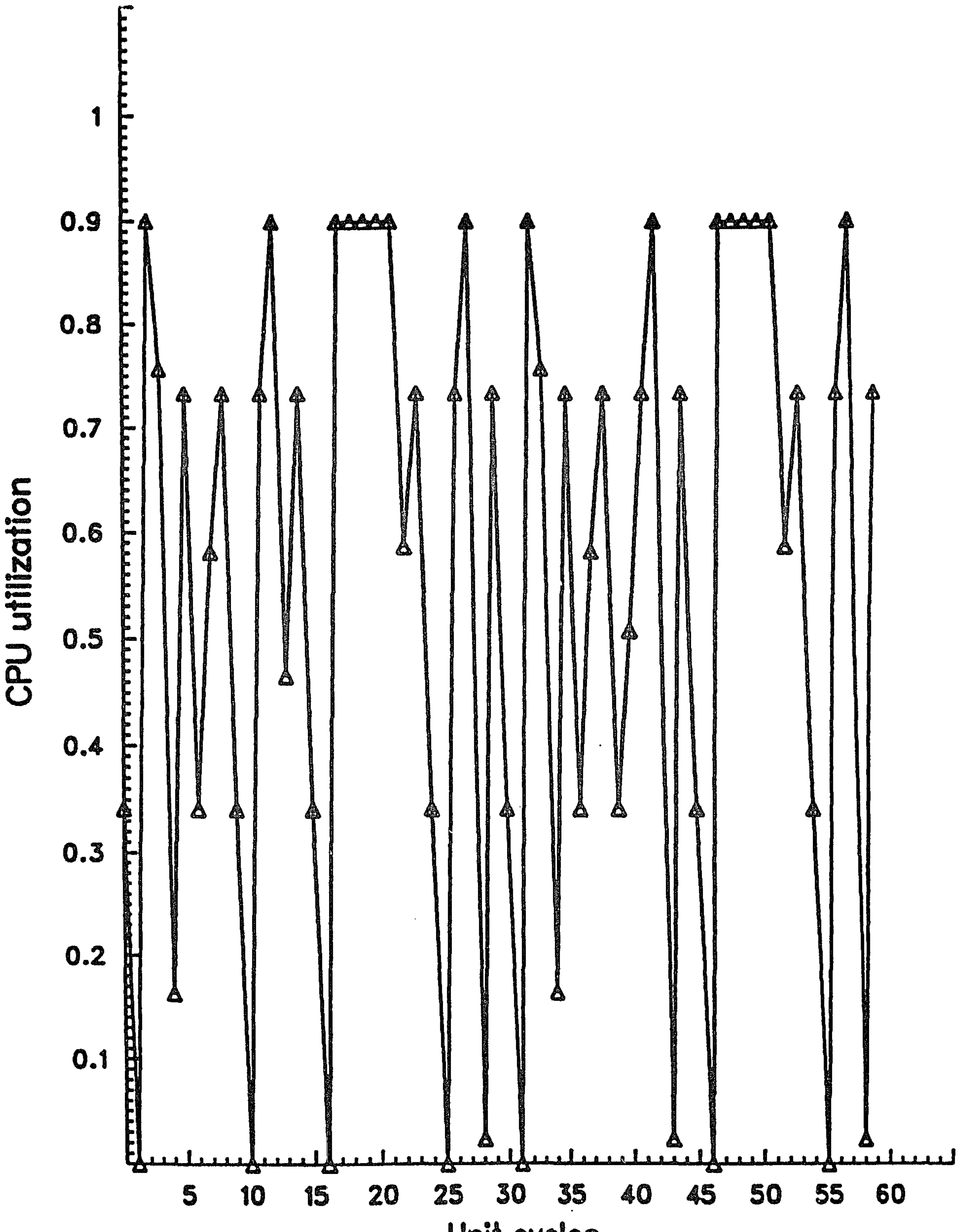


Figure 6.5: CPU utilization with $U = 0.6$ and $R = 0.1$.

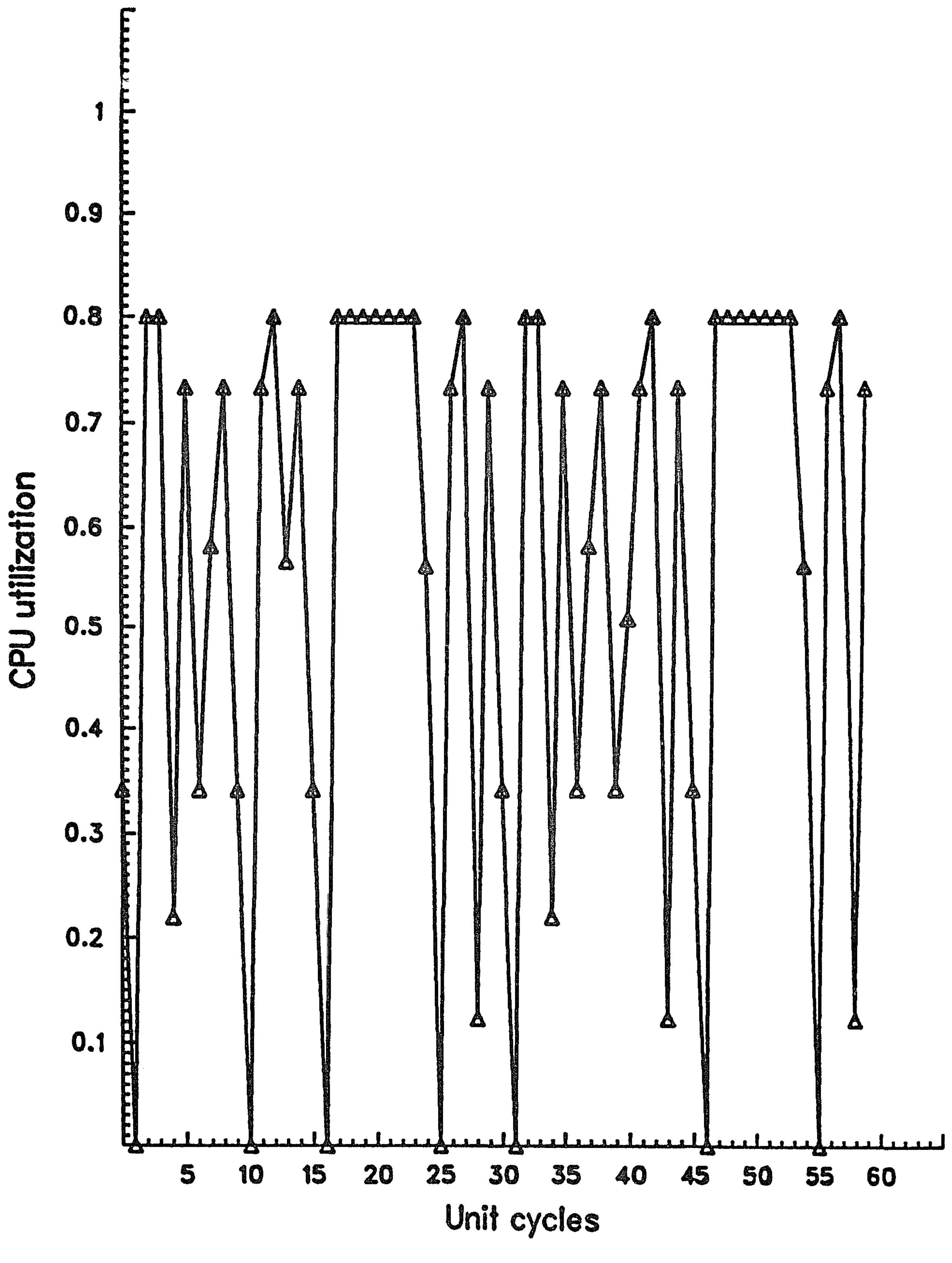


Figure 6.6: CPU utilization with $U = 0.6$ and $R = 0.2$.

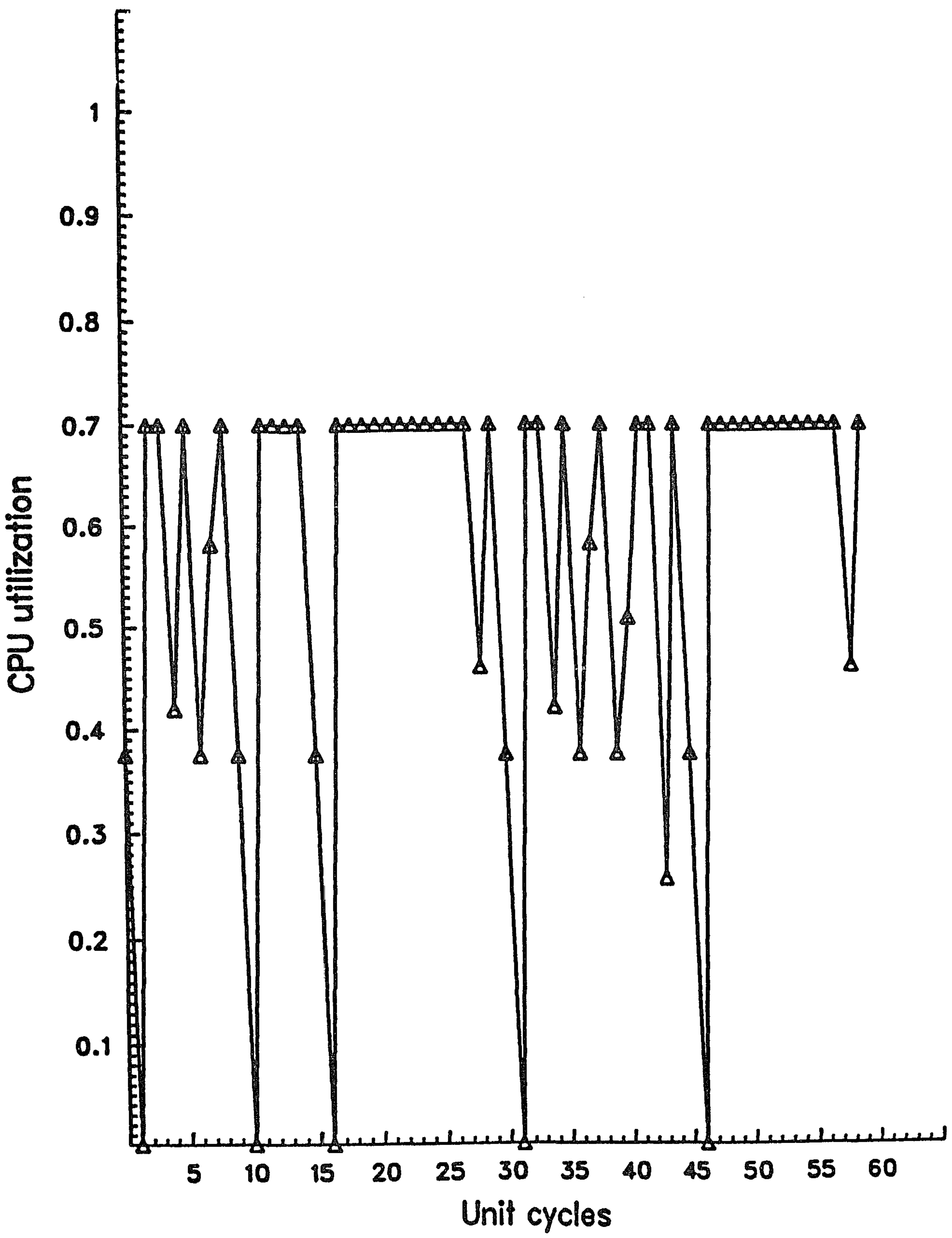


Figure 6.7: CPU utilization with $U = 0.6$ and $R = 0.3$.

where $\overline{F} = \frac{\sum_{i=a_\alpha}^{a_\alpha+D_\alpha} F(i)}{D_\alpha}$. $\overline{F}$ is the average unused CPU time in $[a_\alpha,\, a_\alpha+D_\alpha]$. D_{min} (D_{max}) is the minimal (maximum) task deadline (more on the deadline distribution will be discussed in Section 6.3.4). $P(D)$ is the probability density function of task deadlines.

We are interested in deriving a condition under which $P(C_\alpha \leq \overline{F}\ D_\alpha)$ is maximized. From Eq. (6.3), it is easy to see that the probability of guaranteeing aperiodic tasks is a monotonic increasing function of $\overline{F}$. When U is fixed, reserving a fraction R of each unit cycle will increase the utilization (or decrease $\overline{F}$) in those unit cycles which were under-utilized or even unused as shown in Figs. 6.5–6.7. Since the probability of guaranteeing an aperiodic task is increased (decreased) by increasing (decreasing) $\overline{F}$, the increase in the probability of guaranteeing aperiodic tasks in these unit cycles with increased $\overline{F}$ must be greater than the decrease in the probability of guaranteeing aperiodic tasks in those unit cycles with decreased $\overline{F}$.

Intuitively, the probability of guaranteeing aperiodic tasks is a monotonic increasing function of R. However, this is not clear when a large increase of $\overline{F}$ in a time interval causes small decreases of $\overline{F}$ in many other time intervals. So, we have the following theorem.

Theorem 6.1 *The probability of guaranteeing aperiodic tasks is a monotonic increasing function of R.*

Proof: Let $\overline{F}_1$ be the average unused fraction of CPU time over a time interval, say I_1. If $\overline{F}_1 < R$, after reserving a fraction R of each unit cycle, the average unused fraction of CPU time over I_1 becomes $\overline{F}'_1 = R$. Since U is fixed over a major cycle, increasing the unused fraction of CPU time in I_1 will cause the unused fraction of CPU time to decrease over some other intervals in which the average unused fractions of CPU time were used to be greater than R (before the reservation). Let $I_{j_1}, I_{j_2}, \cdots, I_{j_k}$ be those intervals and $\overline{F}_{j_1}$ ($\overline{F}'_{j_1}$), $\overline{F}_{j_2}$ ($\overline{F}'_{j_2}$), $\cdots$, $\overline{F}_{j_k}$ ($\overline{F}'_{j_k}$) be the average unused fractions of CPU time over these intervals before (after) reserving the CPU time as shown in Fig. 6.8. Note that the distance between I_{j_k} and I_1 is larger for larger k as shown in Fig. 6.8. Then we have

$\overline{F}_1 + \sum_{\ell=1}^{k} \overline{F}_{j_\ell} = \overline{F}'_1 + \sum_{\ell=1}^{k} \overline{F}'_{j_\ell}$. Note that $\overline{F}_{j_\ell} > R$, $\overline{F}'_{j_\ell} = \overline{F}_{j_\ell} - R$ if $\overline{F}_{j_\ell} \geq 2R$, and $\overline{F}_{j_\ell} = R$ if $\overline{F}_{j_\ell} < 2R$ for $\ell = 1, \ldots, k$.

Note that the increase of $\overline{F}'_1$ in I_1 causes many decreases of the unused fraction of CPU time over intervals $I_{j_1}, I_{j_2}, \cdots, I_{j_k}$. Let $\Delta\overline{F}_{j_\ell} = |\overline{F}_{j_\ell} - \overline{F}'_{j_\ell}|$ for $\ell = 1, \ldots k$. Then each of these $\Delta\overline{F}_{j_\ell}$'s will contribute to $\overline{F}'_1$. So, $\Delta\overline{F}_1 = |\overline{F}_1 - \overline{F}'_1| = \sum_{\ell=1}^{k} \Delta\overline{F}_{j_\ell}$. Also, note that the decreases of the unused CPU time over intervals $I_{j_1}, I_{j_2}, \cdots, I_{j_k}$ due to the increase of $\overline{F}'_1$ will occur first on I_{j_1}, then I_{j_2}, and so on, because in the RB scheduling algorithm the task's scheduled time is shifted to an adjacent unit cycle first. So, it is sufficient to consider the following three cases.

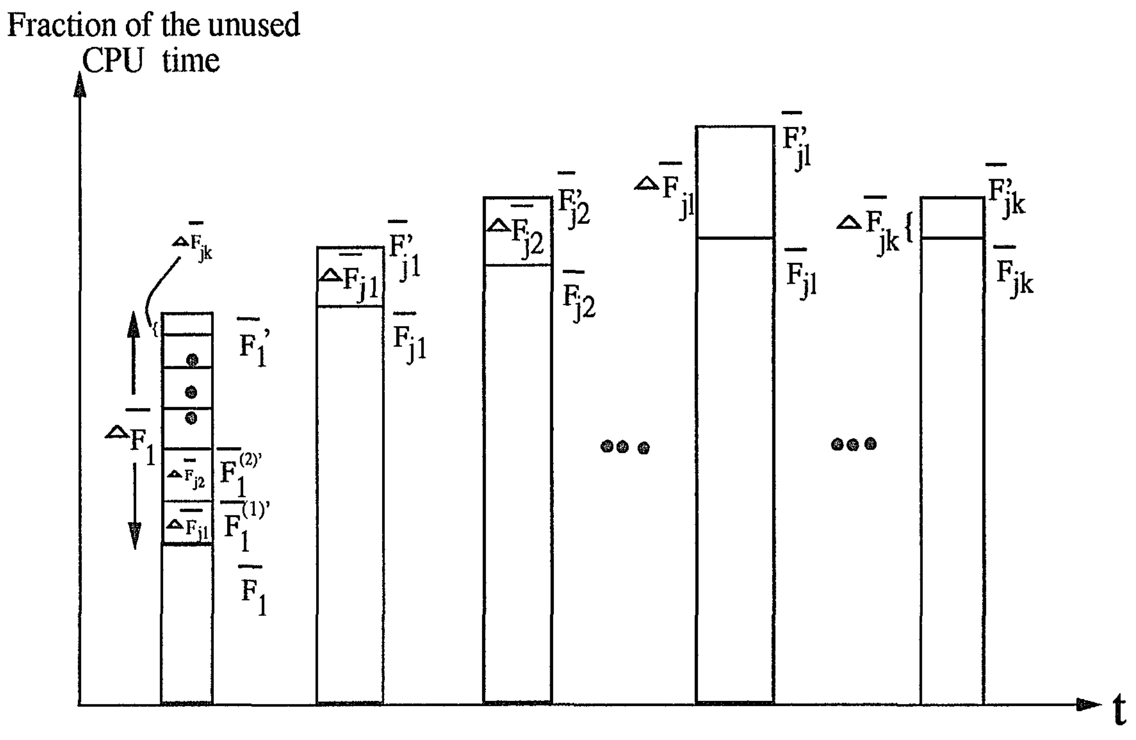


Figure 6.8: The variation of the unused CPU time over some time intervals after reserving a fraction R of CPU time in each unit cycle.

(1). Consider the variation of $\overline{F}_{j_1}$ in I_{j_1} first. For notational convenience, let $\overline{F}_1^{(1)'}$ be the unused CPU time in I_1 when only $\Delta\overline{F}_{j_1}$ is considered, i.e., assuming $\Delta\overline{F}_{j_\ell} = 0$ for $\ell = 2, \ldots, k$. We then have $\overline{F}_1^{(1)'} = \overline{F}_1 + \Delta\overline{F}_{j_1}$. The variation of the probability of guaranteeing an aperiodic task $\nu_\alpha = (a_\alpha, C_\alpha, D_\alpha)$ is computed by:

$$\begin{aligned} P_1(C_\alpha \leq \overline{F}_1\ D_\alpha) - P_1(C_\alpha \leq \overline{F}_1^{(1)'}\ D_\alpha) &= P(D_\alpha)(e^{-\mu \overline{F}_1^{(1)'}\ D_\alpha} - e^{-\mu \overline{F}_1\ D_\alpha}) \\ P_2(C_\alpha \leq \overline{F}_{j_1}\ D_\alpha) - P_2(C_\alpha \leq \overline{F}'_{j_1}\ D_\alpha) &= P(D_\alpha)(e^{-\mu \overline{F}'_{j_1}\ D_\alpha} - e^{-\mu \overline{F}_{j_1}\ D_\alpha}). \end{aligned}$$

The increased probability of guaranteeing the task is $P(D_\alpha)(e^{-\mu \overline{F}_1\ D_\alpha} - e^{-\mu \overline{F}_1^{(1)'}\ D_\alpha})$ while the reduced probability of guaranteeing the task is $P(D_\alpha)(e^{-\mu \overline{F}'_{j_1}\ D_\alpha} - e^{-\mu \overline{F}_{j_1}\ D_\alpha})$. So, the net change is:

$$\begin{aligned} \Delta P^{(1)}(\text{guarantee } \nu_\alpha) &= P(D_\alpha)\left[(e^{-\mu \overline{F}_1\ D_\alpha} - e^{-\mu \overline{F}_1^{(1)'}\ D_\alpha}) + (e^{-\mu \overline{F}_{j_1}\ D_\alpha} - e^{-\mu \overline{F}'_{j_1}\ D_\alpha})\right] \\ &= P(D_\alpha)\left[e^{-\mu \overline{F}_1\ D_\alpha}(1 - e^{-\mu \Delta \overline{F}_{j_1}\ D_\alpha}) + e^{-\mu \overline{F}'_{j_1}\ D_\alpha}(e^{-\mu \Delta \overline{F}_{j_1}\ D_\alpha} - 1)\right] \\ &= P(D_\alpha)(1 - e^{-\mu \Delta \overline{F}_{j_1}\ D_\alpha})(e^{-\mu \overline{F}_1\ D_\alpha} - e^{-\mu \overline{F}'_{j_1}\ D_\alpha}). \end{aligned}$$

From the initial condition we have $\overline{F}_1 < R$ and $\overline{F}'_{j_1} \geq R$, thus $\overline{F}_1 < \overline{F}'_{j_1}$.
So, $\Delta P^{(1)}(\text{guarantee } \nu_\alpha) > 0$.

(2). Consider only the variation of the unused CPU time in intervals I_{j_1} and I_{j_2}. For notational convenience, let $\overline{F}_1^{(2)'} = \overline{F}_1^{(1)'} + \Delta \overline{F}_{j_2}$. The net change in the probability of guaranteeing ν_α is

$$\Delta P^{(2)}(\text{guarantee } \nu_\alpha) = P(D_\alpha)(1 - e^{-\mu \Delta \overline{F}_{j_2}\ D_\alpha})(e^{-\mu \overline{F}_1^{(2)}\ D_\alpha} - e^{-\mu \overline{F}'_{j_2}\ D_\alpha}).$$

Since $\overline{F}_1^{(2)} < \overline{F}'_{j_2}$, $\Delta P^{(2)}(\text{guarantee } \nu_\alpha) > 0$.

(3). Generally, consider the variations of the unused CPU time in intervals I_{j_1} to I_{j_ℓ}, and let $\Delta \overline{F}_{j_\ell} = \overline{F}_{j_\ell} - \overline{F}'_{j_\ell}$ and $\overline{F}_1^{(\ell)'} = \overline{F}_1^{(\ell-1)'} + \Delta \overline{F}_{j_\ell}$. The net change in the probability of guaranteeing ν_α is

$$\Delta P^{(\ell)}(\text{guarantee } \nu_\alpha) = P(D_\alpha)(1 - e^{-\mu \Delta \overline{F}_{j_\ell}\ D_\alpha})(e^{-\mu \overline{F}_1^{(\ell)}\ D_\alpha} - e^{-\mu \overline{F}'_{j_\ell}\ D_\alpha}).$$

Since $\overline{F}_1^{(\ell)} < \overline{F}'_{j_\ell}$, $\Delta P^{(\ell)}(\text{guarantee } \nu_\alpha) > 0$ for $\ell = 1, \ldots, k$.

As a result, the increase in the probability of guaranteeing aperiodic tasks by increasing $\overline{F}$ in I_1 will always be greater than the total reduced probability of guaranteeing aperiodic tasks in $I_{j_1}, \cdots, I_{j_k}$. Since $1-e^{-\mu \Delta\overline{F}_{j_\ell}}$ is a monotonic increasing function of $\Delta\overline{F}_{j_\ell}$, the increase in the probability of guaranteeing aperiodic tasks is a monotonic increasing function of R.

In practice, the decreases of the unused CPU time over some intervals may be caused by the increases of the unused CPU over a multiple number of intervals instead of only one interval I_1 as discussed above. However, we can always identify the decreases of the unused CPU time over those intervals which is caused by the increase of the unused CPU time in a particular interval and apply the same arguments, thus we show that the increase in the probability of guaranteeing aperiodic tasks is a monotonic increasing function of R.

□

6.3.3 Deriving R_{max}

Since Eq. (6.3) is a monotonic increasing function of R, it is desirable to derive R_{max} without missing the deadline of any periodic task. Consider the case of two periodic tasks $\tau_1 = (C_1, T_1)$ and $\tau_2 = (C_2, T_2)$, and let $T_1 < T_2$. As discussed in [12], the critical instant for any task occurs whenever it arrives simultaneously with all other higher priority tasks. So, the most difficult situation for scheduling τ_2 is when it is released at the same time as τ_1. It was shown in [12] that the minimal U occurs when $C_1 = T_2 - T_1\lfloor T_2/T_1 \rfloor$, leading to :

$$U = \frac{\lceil\frac{T_2}{T_1}\rceil C_1 + C_2}{T_2}.$$

So, R_{max} is the maximum unused fraction of CPU time, i.e., .$R_{max} = 1-U = 1-\frac{\lceil\frac{T_2}{T_1}\rceil C_1+C_2}{T_2}$. Generally, R_{max} for a set of m periodic tasks $(C_1, T_1), \cdots, (C_m, T_m)$ such that $T_1 < T_2 < \ldots < T_m$ is

$$R_{max} = 1 - \frac{\sum_{i=2}^{m}\lceil\frac{T_i}{T_1}\rceil C_1 + \sum_{i=3}^{k}\lceil\frac{T_i}{T_2}\rceil C_2 + \ldots + \lceil\frac{T_{(m-1)}}{T_m}\rceil C_m + C_k}{T_m}. \qquad (6.4)$$

6.3.4 Calculating the Probability of Guaranteeing Aperiodic Tasks

Substituting the R_{max} derived from Eq. (6.4) into Eq. (6.3), the probability of guaranteeing aperiodic tasks can be maximized, because $P(C_\alpha \leq \overline{F}\ D_\alpha)$ is a monotonic increasing function of R. Since deadlines of aperiodic tasks are known upon their arrival, $P(D)$ in Eq. (6.3) can be calculated and the probability of guaranteeing aperiodic tasks can then be derived from Eq. (6.3). However, any *a priori* performance evaluation requires the distribution of aperiodic task deadlines. A uniform or exponential distribution of deadlines has been commonly used in early studies [16, 19, 24]. In case of an exponential distribution, randomly-generated deadlines are close to their mean value, and thus the variation of deadlines is not large. The probability of guaranteeing aperiodic tasks is closely related to the distribution of deadlines; the performance data obtained from an exponential distribution strongly depends on the mean value used. Since the distribution of deadlines of aperiodic tasks is application-dependent, one can let

$$D = D_{min} + \lfloor (1+\alpha X)C_i \rfloor, \tag{6.5}$$

where D_{min} is the minimal deadline of an aperiodic task, α is an integer number, and X is a random variable uniformly distributed in [0, 1]. As stated earlier, C_i is assumed to have an exponential distribution with mean μ.

The deadline distribution in Eq. (6.5) allows us to adjust the control parameters, D_{min} and α, to choose an appropriate range of deadlines for aperiodic tasks. D_{min} serves a purpose similar to the minimal separation p in [28]. By varying D_{min}, one can control the worst-case probability of missing deadlines, thus guaranteeing the deadlines up to any acceptable level. $\lfloor (1+\alpha X)C_i \rfloor$ determines the range of deadlines; a large α will generally result in a wide range of distribution. Since it is assumed that X is uniformly distributed and C_i is exponentially distributed, the range of deadlines is a composite function of uniform and exponential distributions. Using Eq. (6.5) to specify the deadline of an aperiodic task can avoid the drawbacks of assuming either a loose bound or a tight bound of deadlines. Moreover, the randomly–generated deadlines are uniformly distributed in the range determined by $\lfloor (1+\alpha X)C_i \rfloor$, so the deadlines is more evenly distributed in a specified range than the exponential distribution. To be consistent with the case of periodic tasks where the deadlines are equal to the period (an integer number of unit cycles), the deadlines of aperiodic tasks are rounded to be integer numbers in Eq. (6.5).

Consider $D_\alpha = D_{min}$ as an example:

$$\begin{aligned} P(D_\alpha = D_{min}) &= P(C_\alpha < \frac{1}{1+\alpha}) + P((1+\alpha X)C_\alpha < 1) P\left(\frac{1}{1+\alpha} \le C_\alpha < 1\right) \\ &= P(C_\alpha < \frac{1}{1+\alpha}) + \frac{1}{\alpha}\left(\frac{1}{C_\alpha} - 1\right) P\left(\frac{1}{1+\alpha} \le C_\alpha < 1\right) \\ &= 1 - e^{-\frac{\mu}{1+\alpha}} + \frac{\mu}{\alpha}\int_{\frac{1}{1+\alpha}}^{1} \frac{e^{-\mu t}}{t} dt + \frac{1}{\alpha}(e^{-\mu} - e^{-\frac{\mu}{1+\alpha}}). \end{aligned} \tag{6.6}$$

If $C_\alpha < \frac{1}{1+\alpha}$, the second term in Eq. (6.5) is always less than 1; this gives the first term in Eq. (6.6). If $1/(1+\alpha) < C_\alpha < 1.0$, D_α is equal to D_{min} when $(1+\alpha X)C_\alpha < 1.0$. Since X is a uniformly distributed random variable in $[0,1]$, $P((1+\alpha X)C_\alpha < 1) = \frac{1}{\alpha}(1/C_\alpha - 1)$, thus giving the second term in Eq. (6.6). If $C_\alpha > 1.0$, D_α is always greater than D_{min}. Similarly, $P(D_\alpha = D_{min} + 1)$ can be shown as:

$$\begin{aligned} P(D_\alpha = D_{min} + 1) &= P((1+\alpha X)C_\alpha \ge 1) P(\frac{1}{1+\alpha} \le C_\alpha < \frac{2}{1+\alpha}) + \\ &\quad P(1 \le (1+\alpha X)C_\alpha < 2) P\left(\frac{2}{1+\alpha} \le C_\alpha < 2\right) \\ &= (1 + \frac{1}{\alpha} - \frac{1}{\alpha t}) \int_{\frac{1}{1+\alpha}}^{\frac{2}{1+\alpha}} \mu e^{-\mu t} dt + \frac{\mu}{\alpha}\int_{\frac{2}{1+\alpha}}^{2} \frac{e^{-\mu t}}{t} dt \qquad (6.7) \\ &= (1 + \frac{1}{\alpha})\left(e^{-\frac{\mu}{1+\alpha}} - e^{-\frac{2\mu}{1+\alpha}}\right) - \frac{\mu}{\alpha}\int_{\frac{1}{1+\alpha}}^{\frac{2}{1+\alpha}} \frac{e^{-\mu t}}{t} dt + \frac{\mu}{\alpha}\int_{\frac{2}{1+\alpha}}^{2} \frac{e^{-\mu t}}{t} dt. \end{aligned}$$

When $C_\alpha \in [\frac{1}{1+\alpha}, \frac{2}{1+\alpha})$, $(1+\alpha X)C_\alpha$ is always less than 2. So, the probability of resulting $D_\alpha = D_{min} + 1$ is $(1+\alpha X)C_\alpha > 1$, thus giving the first term in Eq. (6.7). When $C_\alpha \in [2/(1+\alpha),\ 2)$, $D_\alpha = D_{min} + 1$ if $(1+\alpha X)C_\alpha \in [1, 2)$; this is the second term in Eq. (6.7). When $C_\alpha > 2$, D_α is always greater than $D_{min} + 1$. Generally, for $P(D_\alpha = D_{min} + k)$, $k = 2, \ldots, D_{max}$, we get:

$$P(D_\alpha = D_{min} + k) = P((1+\alpha)C_\alpha \ge k) P(\frac{k}{1+\alpha} \le C_\alpha < \frac{k+1}{1+\alpha}) +$$

$$\begin{aligned} & P(k \leq (1+\alpha)C_\alpha < k+1)P\left(\frac{k+1}{1+\alpha} \leq C_\alpha < k\right) \\ = \; & (1+\frac{1}{\alpha}-\frac{k}{\alpha t})\int_{\frac{k}{1+\alpha}}^{\frac{k+1}{1+\alpha}} \mu e^{-\mu t}dt + \frac{\mu}{\alpha}\int_{\frac{k}{1+\alpha}}^{k} \frac{e^{-\mu t}}{t}dt \qquad (6.8) \\ = \; & (1+\frac{1}{\alpha})\left(e^{-\frac{k\mu}{1+\alpha}} - e^{-\frac{(k+1)\mu}{1+\alpha}}\right) - \frac{k\mu}{\alpha}\int_{\frac{k}{1+\alpha}}^{\frac{k+1}{1+\alpha}} \frac{e^{-\mu t}}{t}dt + \frac{\mu}{\alpha}\int_{\frac{i}{1+\alpha}}^{k} \frac{e^{-\mu t}}{t}dt. \end{aligned}$$

From Eq. (6.5), the deadline of an aperiodic task is generated based on its computation time which was randomly–generated from an exponential distribution. The effect of D_{min} and α on the probability of guaranteeing an aperiodic task will be discussed in the next section.

6.4 Implementation and Performance Analysis of RB Algorithm

To implement the RB algorithm in a real–world system, one needs to consider the issues of both local and global scheduling (load sharing) of aperiodic tasks.

6.4.1 Local Scheduling

Periodic tasks are considered as the base workload and assigned higher priority than aperiodic tasks. The U calculated from Eq. (6.1) is the CPU utilization for a given set of periodic tasks. Aperiodic tasks are processed by utilizing the unused fraction of CPU time, $1-U$. The decision of accepting an aperiodic task $= (a_\alpha, C_\alpha, D_\alpha)$ is made by comparing C_α with $\overline{F}D_\alpha$. If $C_\alpha \leq \overline{F}D_\alpha$, this task will be accepted and scheduled locally; otherwise it will be either transferred out or rejected. Transferring an aperiodic task is the subject of global scheduling or load sharing and will be discussed in the following subsection. Since acceptance of an aperiodic task depends on the amount of unused fraction of CPU time on a node, if U is high, say over 0.6, the node may not be able to process many aperiodic tasks. In such a case, the chance of accepting more than two aperiodic tasks in a short time interval is very small, so the FCFS scheduling policy can be used. However, if U is low, say below 0.4, the node will be able to process a large number of aperiodic tasks, and more than two aperiodic tasks are likely to be accepted within a short time interval. Thus, a good local scheduling scheme better than the FCFS policy is desirable to maximize the probability of guaranteeing aperiodic tasks.

Hong *et al.* has proved that the minimum laxity first (MLF) policy is the optimal local scheduling scheme in real-time systems without periodic tasks [24]. Unfortunately, MLF policy is not feasible in the RB algorithm for the following reason. Since periodic tasks are considered as the base workload, $U(i)$ varies even when some fraction R of a unit cycle is reserved as shown in Fig. 6.7. So, the laxity of an aperiodic task cannot be given *a priori* and must be calculated upon its arrival. Moreover, the laxity of an aperiodic task depends on its arrival time. The difficulty of using the MLF policy for the RB scheme can be seen from the following scenario. Consider an aperiodic task $\nu_1 = (a, C_1, D_1)$. If ν_1 is accepted and scheduled locally, $U(i)$ in $[s_a, s_{a+D_1}]$ will be updated accordingly. If a second aperiodic task ν_2 arrives later, say at $t = a + 1$, its laxity can be calculated based on either the updated $U(i)$, or the original $U(i)$. If it is calculated based on the updated $U(i)$, the MLF policy does not make any sense, because ν_1 has already been scheduled. If it is calculated based on the original $U(i)$ and scheduled ahead of ν_1 when its laxity is smaller than ν_1's original laxity, ν_1's laxity needs to be re-calculated based on the newly updated $U(i)$ after scheduling ν_2. It may turn out that the adjusted laxity for ν_1 may be shorter than ν_2's laxity leading to a conflict. In fact, it is also shown in [24] that the MLF policy works best only in nonpreemptive systems [24] to avoid the above difficulty. Since in our model, all tasks can be preempted at any time, the MLF policy is not feasible for the RB scheme.

The earliest deadline (ED) policy does not outperform the FCFS policy in the RB scheme, because if the ED policy can guarantee a number of aperiodic tasks, so can the FCFS policy for the RB scheme for the following reason. Consider two aperiodic tasks $\nu_1 = (a, C_1, D_1)$, $\nu_2 = (a, C_2, D_2)$, and assume $D_1 < D_2$. Suppose these two aperiodic tasks can both be guaranteed by the ED policy. Then the sum of the unused CPU time in the unit cycles from s_a to s_{a+D_2} will be at least $C_1 + C_2$. If ν_2 is scheduled before ν_1, one can reserve at least C_1 of CPU time from s_a to s_{a+D_1} while still meeting the deadline of ν_2. Since the sum of the unused CPU time is at least C_1 from s_a to s_{a+D_1}, ν_1 can also be guaranteed even if it is scheduled after ν_2, that is, the FCFS scheduling policy can also guarantee both ν_1 and ν_2.

Generally, consider a set of aperiodic tasks $\nu_1 = (a, C_1, D_1)$, $\nu_2 = (a, C_2, D_2)$, $\ldots$, $\nu_n = (a, C_n, D_n)$. For convenience, assume $D_1 \leq D_2 \cdots \leq D_n$. If these tasks can be guaranteed by the ED scheduling algorithm, we have $\sum_{i=1}^{k} C_i \leq \sum_{j=a}^{a+D_k} F_j$ for $k = 1, \ldots, n$. When these tasks arrive in any arbitrary order, they can still be guaranteed by the FCFS scheduling algorithm. For example, suppose $\nu_m = (a, C_m, D_m)$ is scheduled first. Since the sum of the unused CPU time from s_a to s_{a+D_m} is at least $\sum_{i=1}^{m} C_i$, we can reserve $\sum_{i=1}^{m-1} C_i$ amount of CPU time from s_a to s_{a+D_m}, such that tasks ν_1 to ν_{m-1} can still be guaranteed even if they are scheduled after ν_m. Similarly, any other task can be scheduled next to ν_m by reserving some amount of CPU time for the rest of tasks with shorter deadlines. So, the FCFS scheduling algorithm can perform as well as the ED policy.

6.4.2 Global Scheduling

As shown in Figs. 6.5 to 6.7, $U(i)$ varies from 0 to $1 - R$. Those aperiodic tasks arriving at the unit cycles with larger $U(i)$ will not be guaranteed with the same probability as those arriving at the unit cycles with smaller $U(i)$. One way to alleviate this problem is to transfer the tasks that cannot be guaranteed locally to some other nodes with a sufficient amount of unused CPU time.

The results in Section 2.5 showed that the LSMSCB can significantly reduce the probability of missing task deadlines. Thus, we want to modify the LSMSCB to implement the RB scheduling algorithm in a distributed system. Note that periodic tasks on each node cannot be transferred, because missing the deadline of a periodic task may mean missing deadlines of all invocations of this task. So, only aperiodic tasks can be transferred for the purpose of load sharing.

It takes t unit cycles to transfer an aperiodic task. The set of periodic tasks and $U(i)$ of each node are known to all other nodes. A node will broadcast its state change to the nodes in its buddy set only when an aperiodic task is scheduled on it; this task can be its own arrival or a transferred-in task. Recall that the transfer policy is to determine when to transfer an aperiodic task and the location policy is to determine where to send the task.

Whenever $C_\alpha > \sum_{a_\alpha}^{a_\alpha+D_\alpha} F(i)$ holds for an aperiodic task $\nu_\alpha = (a_\alpha, C_\alpha, D_\alpha)$, the task must be transferred. A node will check $U(i)$ of the nodes in its buddy set according to its preferred list, and the first node that satisfies the condition $C_\alpha \leq \sum_{a_\alpha+tu}^{a_\alpha+D_\alpha} F(i)$ will be selected as the receiver for the task. Since the time to transfer a task is $t\,u$, the transferred task will arrive at the receiver node at time $a_\alpha + t\,u$, assuming that both the location and transfer policies will take no time. So, the minimal deadline of this task is assumed to be $t\,u + C_\alpha$ relative to a_α.

The performance of the RB scheme is evaluated by simulation and compared with the analytic results. Several interesting results are obtained. First, as shown in Fig. 6.9, the probability of missing aperiodic task deadlines decreases as R increases, confirming Theorem 6.1. Increasing the value of α reduces the probability of missing aperiodic task deadlines. The effect of increasing D_{min} for aperiodic tasks can be seen in Figs. 6.10 and 6.11. The probability of missing a deadline can be reduced to as small as 10^{-4} with $D_{min} = 5$. The effects of varying D_{min} and R on the probability of missing deadlines are plotted in Fig. 6.12. The effects of varying U are plotted in Fig. 6.13. The probability of missing deadlines of aperiodic tasks increases as U increases. Thus, U must be kept below a certain level in order to reduce the probability of missing aperiodic task deadlines to an acceptable level.

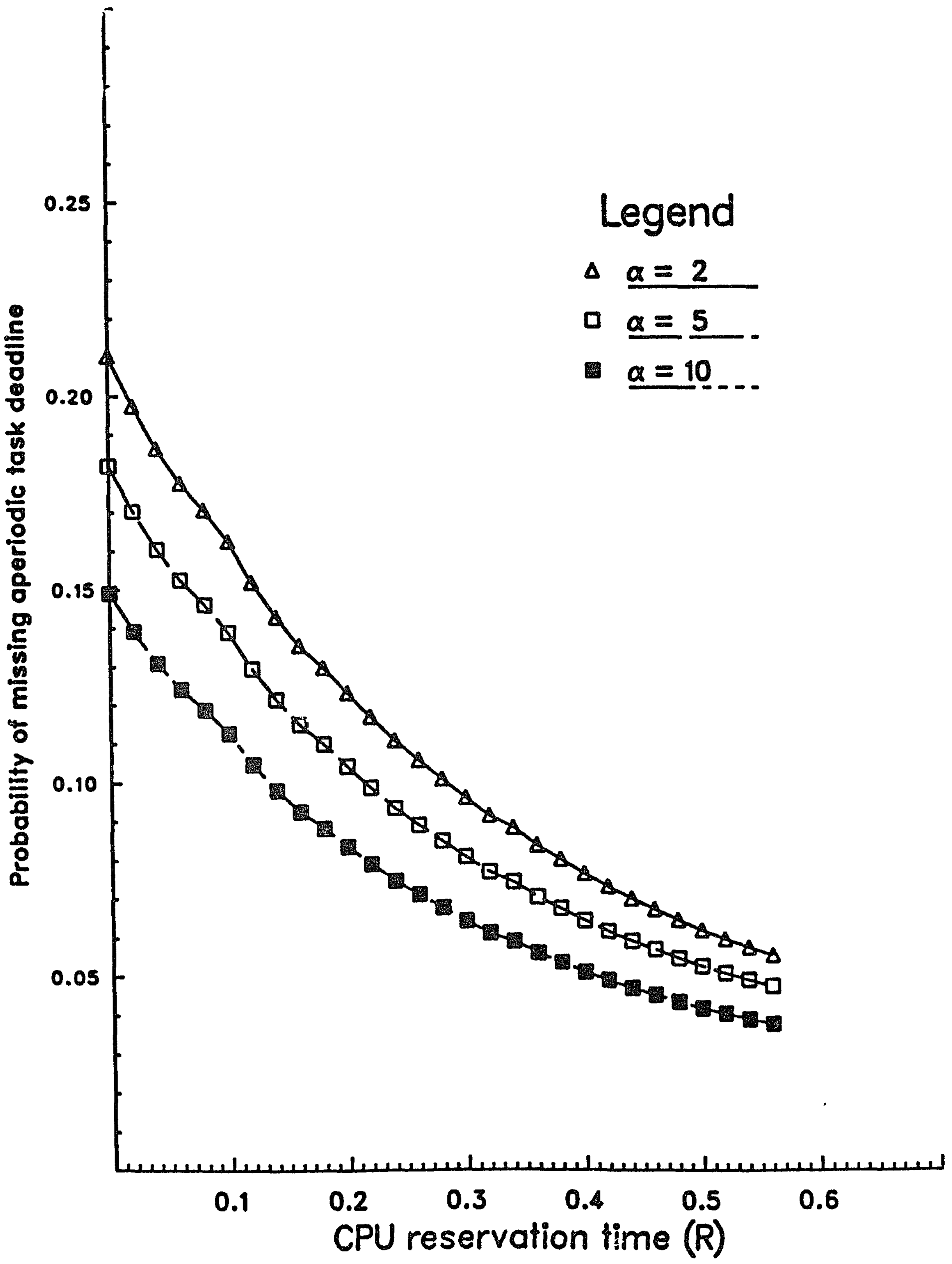


Figure 6.9: Probability of missing aperiodic-task deadlines vs. CPU reservation ($U = 0.4$, $D_{min} = 2$).

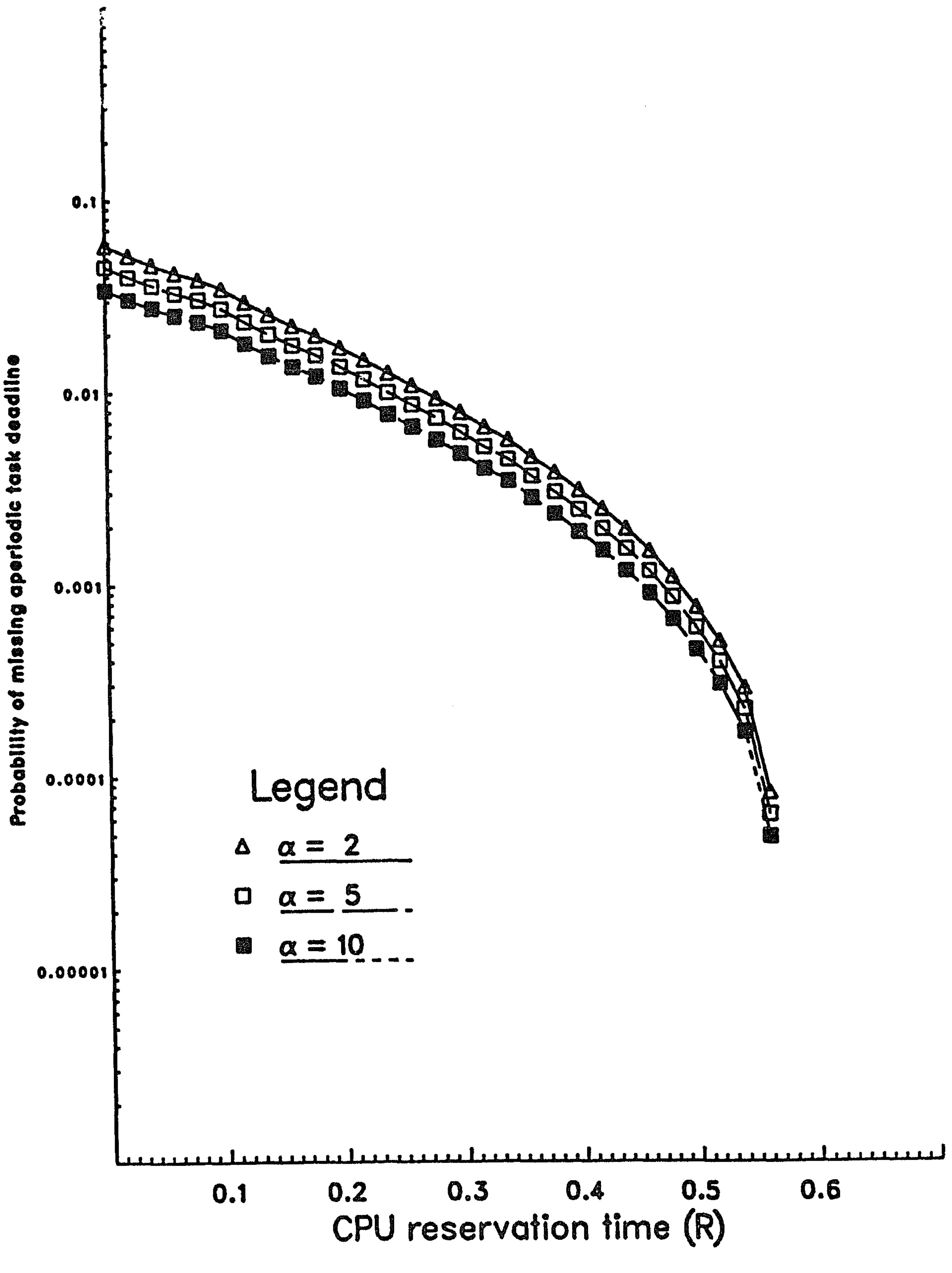


Figure 6.10: Probability of missing aperiodic-task deadlines vs. CPU reservation ($U = 0.4$, $D_{min} = 5$).

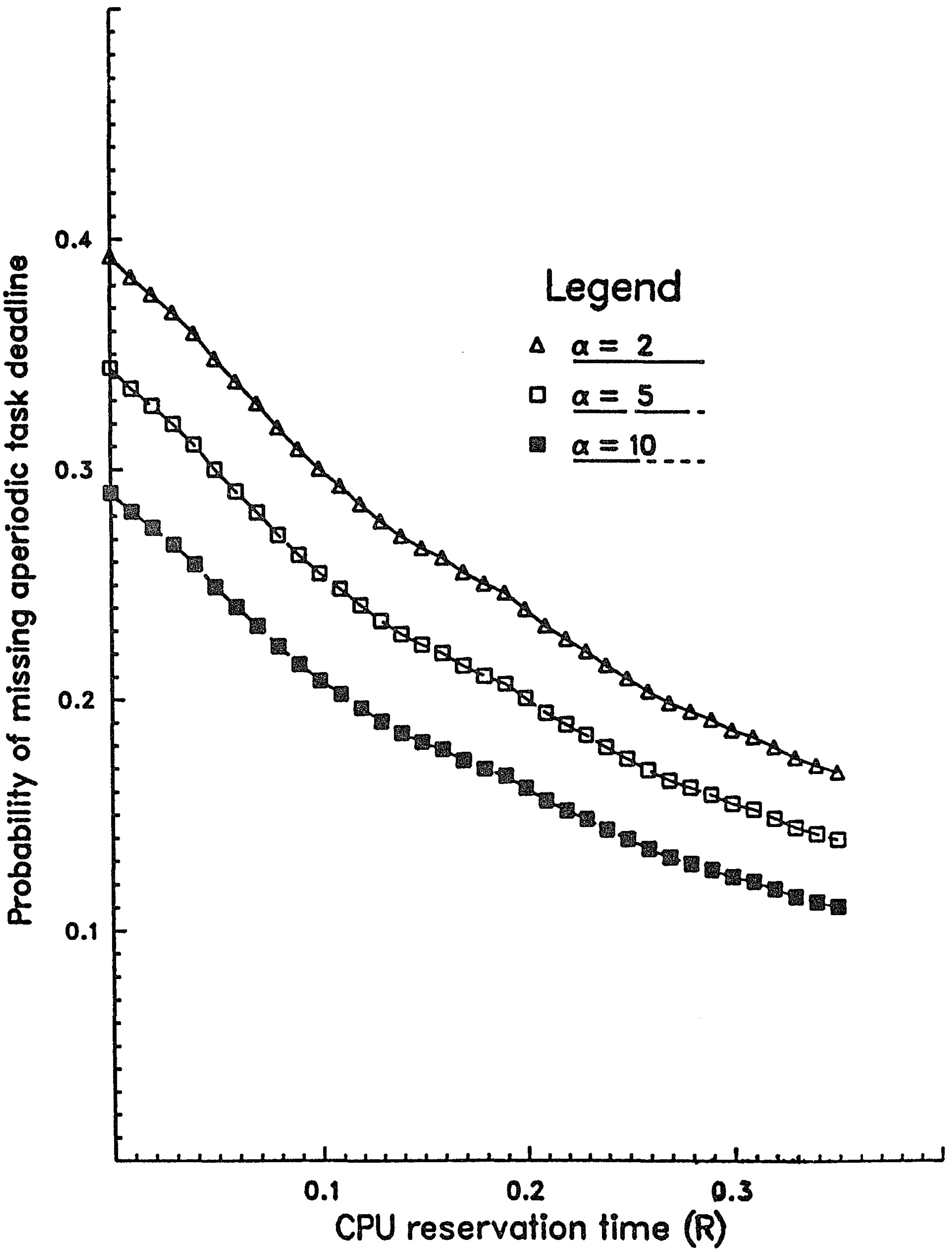


Figure 6.11: Probability of missing aperiodic-task deadlines vs. CPU reservation ($U = 0.6$, $D_{min} = 2$).

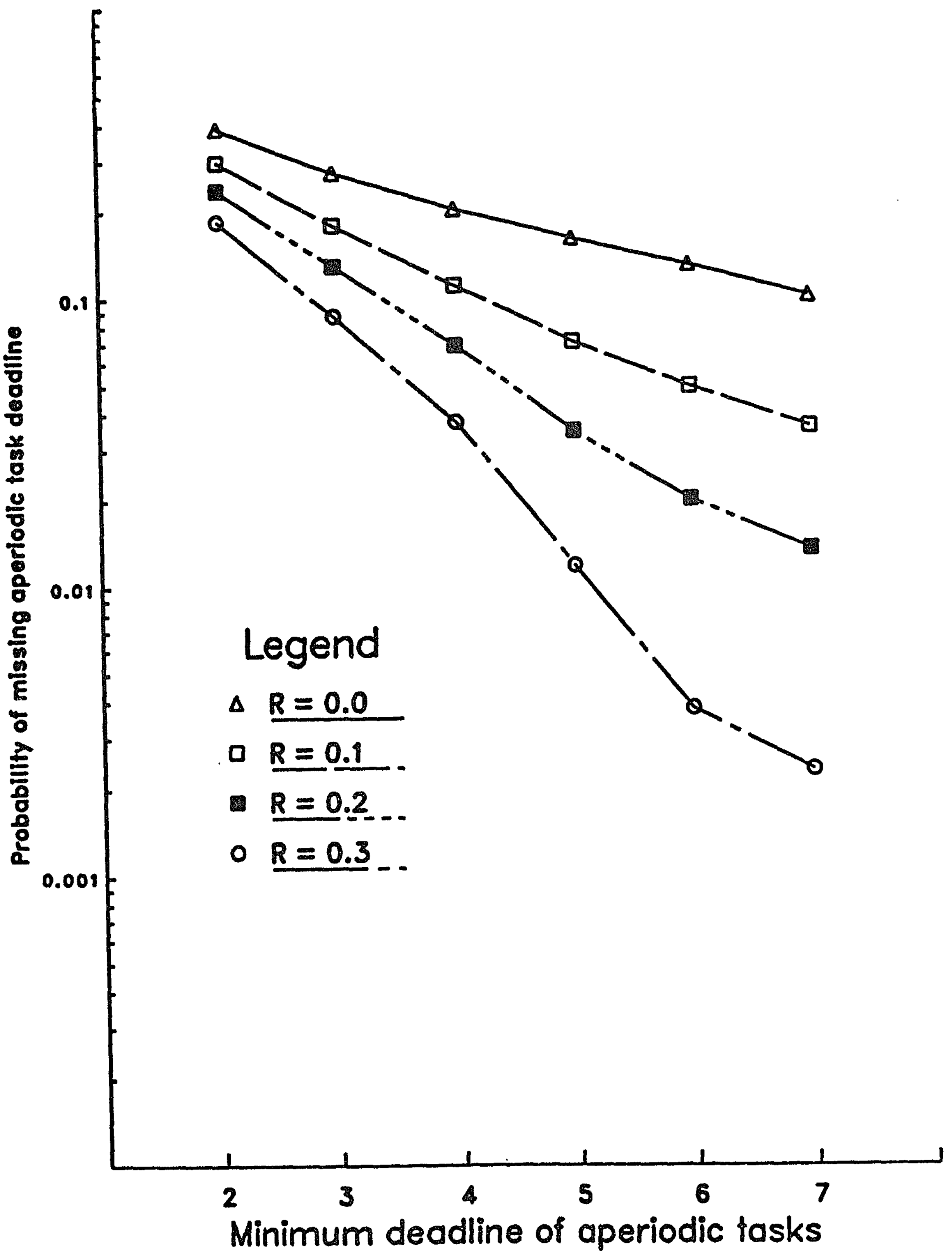


Figure 6.12: Probability of missing aperiodic-task deadlines vs. minimum deadline ($U = 0.6$, $\alpha = 2$).

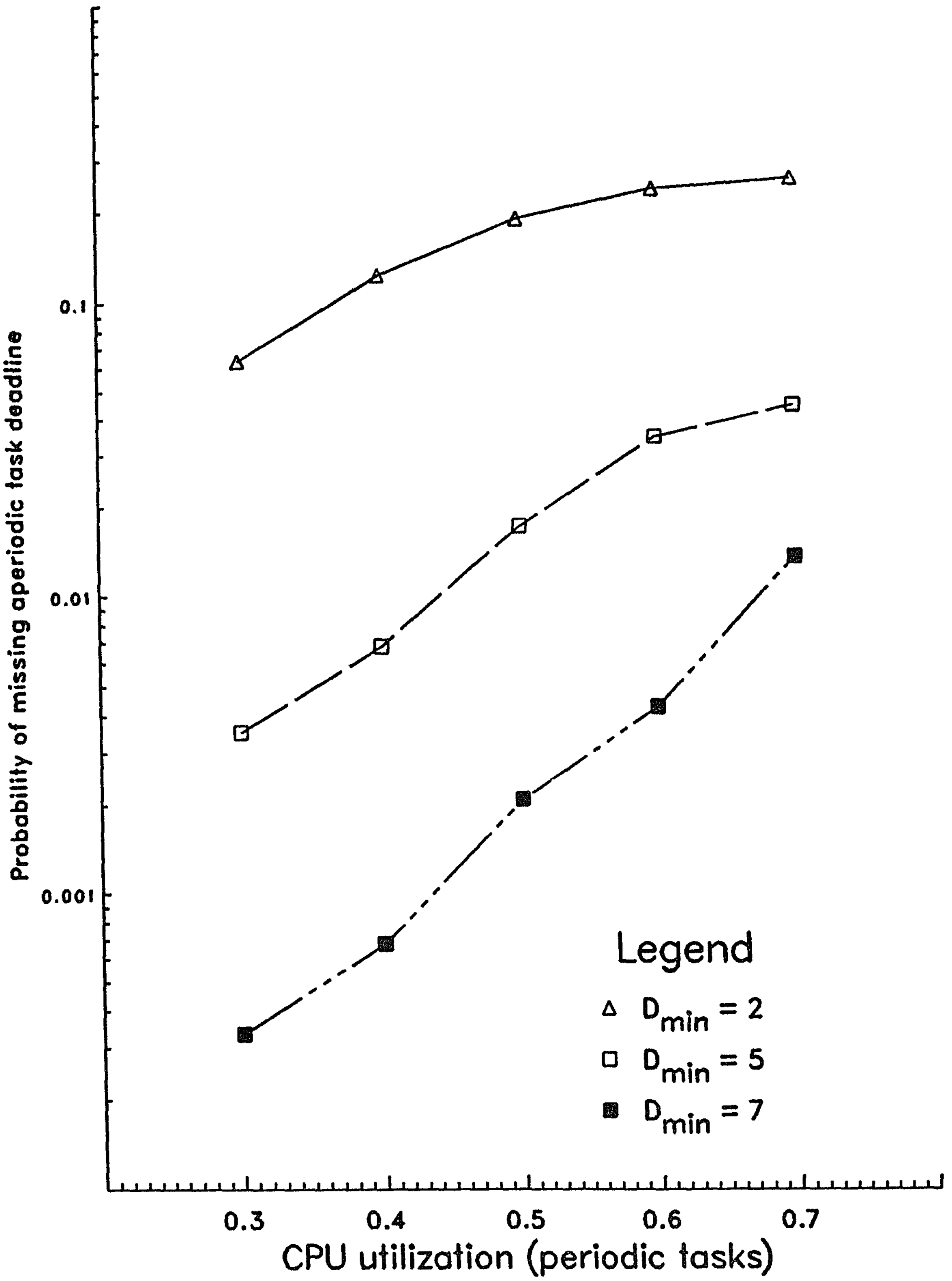


Figure 6.13: Probability of missing aperiodic-task deadlines vs. CPU utilization ($R = 0.2$, $\alpha = 2$).

CHAPTER 7

CONCLUSIONS

7.1 Summary

This dissertation has addressed the problem of scheduling both periodic and aperiodic tasks in distributed real–time systems. One of the major contributions in this dissertation is the LSMSCB for aperiodic tasks by which the probability of missing task deadlines can be reduced to below 10^{-7} in most cases. Using the buddy sets and preferred lists to solve the coordination and congestion problems in distributed systems is the second contribution. The third contribution is the reservation–based scheduling algorithm which can guarantee the periodic tasks while minimizing the probability of missing the deadlines of aperiodic tasks. Specifically, the contributions of each chapter are summarized as follows.

- In Chapter 2, the LSMSCB is developed for scheduling aperiodic tasks in distributed real–time systems. The performance of the LSMSCB is analyzed by using an embedded Markov chain model. Numerical solutions are derived from the model equations and provide many interesting results for the design of threshold patterns and buddy sets. It is shown that by selecting an appropriate threshold pattern and a buddy set, P_{dyn} can be reduced to below 10^{-7} for a wide range of system load, thus making the LSMSCB suitable for various real–time applications.

  The LSMSCB is based on an interesting but different idea from existing approaches which attempt to modify/generalize a periodic–task scheduling algorithm for aperiodic tasks [74]. Instead of trying to design a complex scheduling algorithm to guarantee aperiodic tasks, tasks arriving at each node are processed on a FCFS basis in the LSMSCB. However, if a node cannot meet the deadline of some tasks, the node will transfer these tasks to the nodes in its buddy set for processing, so deadlines can be met by using the 'combined' processing power of all nodes in the system. Both simulation and analytic results indicate that the LSMSCB is very efficient in meeting the deadline of aperiodic tasks.

- An approximate closed–form expression for the distribution of QL under the LSMSCB is derived in Chapter 3. Based on the closed–form distribution, 'optimal' threshold patterns and buddy set sizes are determined either by minimizing the communication overhead — such as the frequency of collecting state information and the number of task transfers — incurred by the LSMSCB while keeping P_{dyn} below a specified level, or by minimizing P_{dyn} while keeping communication cost below a given level. An upper bound of processor utilization is also derived while reducing P_{dyn} to a given level. Moreover, the system utilization is significantly improved, as compared to the results calculated based on the upper bound model in Section 2.5.

- In Chapter 4, the location policy used in the LSMSCB is elaborated on for hypercube multicomputers. The coordination and congestion problems associated with the use of preferred lists and buddy sets are identified and solved. The performance of the proposed location policy is analyzed and compared with others, such as probing, random selection, and bidding algorithms, in terms of the number of task collisions. The proposed location policy is shown to minimize the number of task collisions, and its superiority to other location policies becomes more pronounced when one or more hot regions are formed in the hypercube.

- Chapter 5 addresses and analyzes two important fault–tolerant issues associated with the LSMSCB in hypercube multicomputers: (i) ordering fault–free nodes as the preferred receivers of overflow tasks and (ii) developing a LS mechanism to handle node failures. Since the occurrence of node failures will destroy the original structure of the preferred lists, three algorithms are proposed to adjust the preferred lists. The preferred lists modified by the proposed algorithms are shown to retain their original properties regardless of the number of faulty nodes in the system. Moreover, these algorithms can either minimize the number of adjustments or minimize the distance between a node and the nodes in its buddy set. A simple backup queue can be implemented based on these algorithms. The communication overhead and delay for maintaining/updating the backup queue are shown to be minimal, thus reducing the number of lost tasks.

- The problem of scheduling both periodic and aperiodic tasks is considered and evaluated in Chapter 6. A reservation–based (RB) scheduling algorithm is proposed to schedule both periodic and aperiodic tasks on distributed real–time systems. The deadlines of periodic tasks are guaranteed first by using the rate monotonic scheduling algorithm. Aperiodic tasks are then scheduled by reserving a fraction, R, of CPU time in each unit cycle. R is shown to have great effects on the probability of guaranteeing aperiodic tasks even when the average utilization is fixed. The relation between R and the probability of guaranteeing an aperiodic task is established, and the maximum feasible value of R is derived under which the probability of guaranteeing an aperiodic task can be maximized. Moreover, the RB scheduling algorithm can be implemented in a distributed real–time system by using the LSMSCB. Both simulation and analysis results indicate that the drop-out rate of aperiodic tasks can be reduced to below 10^{-4}, whereas the previous best results were shown to be 0.05 [19].

7.2 Future Work

Although the LSMSCB is shown to greatly improve the performance of a distributed real–time system, there are several remaining issues that warrant further investigation.

- All tasks are assumed to have an identical, or identically–distributed, execution times in Chapter 2, which is not realistic. When all tasks have an identical execution time, QL can be used to determine the workload of each node as was discussed in Chapter 2. If tasks have different execution times, cumulative execution time (CET), rather than QL, should be used to measure the workload in each node [75]. Moreover, task execution time may be a random variable due to, for example, conditional branches and loops in the task. Modeling the LSMSCB in the case of general task execution times is an interesting problem. For example, the load states, U, F, and V used in the LSMSCB will no longer be meaningful, because the task deadline must be compared with the CET on a node to determine whether the task can be transferred to that node. One way to solve this problem is to use multiple thresholds as discussed in [75]. In [75], k thresholds are used to determine the load states on a node and the node will broadcast the state change to all other nodes when its load state switches from an odd–numbered threshold interval to an even–numbered threshold interval. Although using multiple thresholds can define the load state more accurately and improve deadline guarantees, the frequency of state change will be higher than the approach of using three thresholds in the LSMSCB. How to define the load state accurately while keeping the communication overhead below an accepted level is an interesting problem.

- The FCFS policy is used as the local scheduling policy in the LSMSCB. Under this policy, if the deadline of a newly arrived task cannot be guaranteed, it will be transferred to another node. However, it is possible that the deadline of the newly arrived task can be guaranteed locally by rearranging the tasks in the queue. For example, suppose the deadline and computation time of the newly arrived task is D_1 and C_1, and the cumulative execution time in the task queue is CET. If $\exists\, \nu_2 = (D_2, C_2)$ in the queue such that $D_2 \geq CET + C_1$ and $D_1 \geq CET'$, where CET' is the cumulative execution time of the tasks queued ahead of ν_2, then replacing ν_2 by ν_1 will be able to guarantee the deadlines of both tasks, thus eliminating one task transfer. Moreover, if the newly arrived task cannot be guaranteed by any possible rearrangement of the schedule, it is still not necessary to transfer the newly arrived task. For example, suppose the deadline of ν_1 can not be guaranteed by node N_i. If the transfer time of ν_1 is too long or its deadline is very short, it might be possible to transfer some other tasks, say ν_2, in the queue with a shorter transfer time or longer deadline such that the deadline of ν_1 can be guaranteed after transferring ν_2. Hence, a good local scheduling algorithm will be able to improve the probability of guaranteeing task deadlines.

- The preferred lists are generated according to the physical distance between nodes, and the nodes with shorter distance between them will be assigned higher preference. When some nodes failed, as was discussed in Chapter 5, the distance between nodes might be changed. How to modify the preferred lists to adapt to this change needs to be studied further. Moreover, P_{dyn} in the presence of faulty nodes is found to be higher than the case without faulty nodes even when some BKQs are provided. This problem occurs when a node fails while some tasks are being processed. If intermediate results are not saved, the tasks must be restarted again, thus increasing the processing time and missing the deadlines. How to design and use BKQs for real–time applications is worth further investigation.

BIBLIOGRAPHY

[1] K. G. Shin, C. M. Krishna, and Y.-H. Lee, "A unified method for evaluating real–time computer controllers and its application," *IEEE Trans. on Auto. Contr.*, vol. AC–30, no. 4, pp. 357–366, April 1985.

[2] D. W. Leinbaugh, "Guaranteed response times in a hard–real–time environment," *IEEE Trans. Software Engr.*, vol. SE-6, no. 1, pp. 85–93, January 1980.

[3] C. M. Krishna, K. G. Shin, and I. S. Bhandari, "Processor tradeoffs in distributed real–time systems," *IEEE Trans. Comput.*, vol. C-36, no. 9, pp. 1030–1040, September 1987.

[4] D. W. Leinbaugh and M. Yamini, "Guaranteed response times in a distributed hard real–time environment," *IEEE Trans. Software Engr.*, vol. SE-12, no. 12, pp. 1139–1144, December 1986.

[5] J. F. Kurose, S. Singh, and R. Chipalkatti, "A study of qusai–dynamic load sharing in soft real–time distributed computer systems," *Proc. of 7th IEEE Real–Time Systems Symposium*, pp. 201–208, 1986.

[6] P. Ma, E. Lee, and M. Tsuchiya, "A task allocation model for distributed computing systems," *IEEE Trans. Computers*, vol. C-31, no. 1, pp. 41–48, January 1982.

[7] C. E. Houstis, "Module allocation of real–time applications to distributed systems," *IEEE Trans. Software Engineering*, vol. SE-16, no. 7, pp. 699–709, 1990.

[8] A. K. Ezzat, R. D. Bergeron, and J. L. Pokoski, "Task allocation heuristics for distributed computing systems," *Proc. of 6th IEEE Int'l. Conf. on Dist. Comp. systems*, pp. 337–346, 1986.

[9] J. Stankovic and K. Ramamritham, "Evaluation of a flexible task scheduling algorithms for distributed hard real–time systems," *IEEE Trans. Computers*, vol. C-34, no. 12, pp. 1130–1144, December 1985.

[10] M. J. Gonzalez and B. W. Jordan, "A framework for the quantitative evaluation of distributed computing systems," *IEEE Trans. Comput.*, vol. C-29, no. 12, pp. 1087–1094, December 1980.

[11] R. Gerber and I. Lee, "Communicating shared resources: A model for distributed real–time systems," *Proc. of 10th Real–time System Symposium*, pp. 68–78, 1989.

[12] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real–time environment," *Journal of ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

[13] R. L. Graham *et al.*, "Optimization and approximation in deterministic sequencing and scheduling: A survey," *Ann. Discrete Math.*, vol. 5, pp. 287–326, 1979.

[14] D. Peng and K. G. Shin, "Modeling of concurrent task execution in a distributed system for real–time control," *IEEE Trans. Comput.*, vol. C-36, no. 4, pp. 500–516, April 1987.

[15] H. Kasahara and S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," *IEEE Trans. Comput.*, vol. C-33, no. 11, pp. 1023–1029, November 1984.

[16] J. A. Stankovic, K. Ramamritham, and S. Cheng, "Evaluation of a flexible task scheduling algorithm for distributed hard real–time systems," *IEEE Trans. Comput.*, vol. C-34, no. 12, pp. 1130–1143, December 1985.

[17] W. Zhao, K. Ramamritham, and J. A. Stankovic, "Scheduling tasks with resource requirements in hard real–time systems," *IEEE Trans. Software Engineering*, vol. SE-13, no. 5, pp. 564–577, May 1987.

[18] J. W. S. Liu, K. J. Lin, and S. Natarajan, "Scheduling real–time, periodic jobs using imprecise results," *Proc. of 8th Real-time System Symposium*, pp. 252–260, 1987.

[19] K. Ramamritham, J. A. Stankovic, and W. Zhao, "Distributed scheduling of tasks with deadlines and resource requirements," *IEEE Trans. Comput.*, vol. C-38, no. 8, pp. 1110–1123, August 1989.

[20] S. R. Biyabani and J. A. Stankovic, "The integration of deadline and criticalness in hard real–time scheduling," *Proc. of 9th Real-time System Symposium*, pp. 152–160, 1988.

[21] C. Shen, K. Ramamritham, and J. A. Stankovic, "Resource reclaiming in real–time," *Proc. of 10th Real-time System Symposium*, pp. 41–50, 1989.

[22] W. Zhao, K. Ramamritham, and J. A. Stankovic, "Scheduling tasks with resource requirements in hard real–time systems," *IEEE Trans. Software Engr.*, vol. SE-13, no. 5, , May 1987.

[23] S. S. Panwar, D. Towsley, and J. K. Wolf, "Optimal scheduling policies for a class of queues with customer deadlines to the beginning of service," *J. ACM*, vol. 35, no. 4, pp. 832–844, October 1988.

[24] J. Hong, X. Tan, and D. Towsley, "A performance analysis of minimum laxity and earliest deadline scheduling in a real–time system," *IEEE Trans. Comput.*, vol. C-38, no. 12, pp. 1736–1744, December 1989.

[25] K. G. Shin and Y.-C. Chang, "Load sharing in distributed real-time systems with state change braodcasts," *IEEE Trans. Comput.*, vol. C–38, no. 8, pp. 1124–1142, August 1989.

[26] K. G. Shin and Y.-C. Chang, "Load sharing in hypercube multicomputers for real-time applications," *Proc. of* 4^{th} *Conf. on Hypercube Concurrent Computers and Applications*, pp. 617–621, March 1989.

[27] T. L. Casavant and J. G. Huhl, "A formal model of distributed decision–making and its application to distributed load balancing," *Proc. of 6th IEEE Int'l. Conf. on Dist. Comp. Systems*, pp. 232–239, 1986.

[28] A. Mok, "Fundamental design problems of distributed systems for the hard real–time environment,", Ph.D. Thesis, MIT, 1983.

[29] J. P. Lehoczky, L. Sha, and J. K. Strosnider, "Enhanced aperiodic resdponsiveness in hard real–time environments," *Proc. of 8th Real–time System Symposium*, pp. 261–270, 1987.

[30] B. Sprunt, J. Lehoczky, and L. Sha, "Exploiting unused periodic time for aperiodic service using the extened priority exchange algorithm," *Proc. of 9th Real–time System Symposium*, pp. 251–258, 1988.

[31] K. Jeffay, R. Anderson, and C. Martel, "On optimal, non-preemptive shceduling of periodic and sporadic tasks," *Technical Report TR-90-019,* Department of Computer Science, The University of North Carolina at Chapel Hill, 1990.

[32] D. J. Farber, J. Feldman, F. R. Heinrich, M. D. Hopwood, K. C. Larson, D. C. Loomis, and L. A. Rowe, "The distributed computing system," *IEEE COMPCON Spring*, pp. 31–34, 1973.

[33] R. G. Smith, "The contract net protocal: high-level communication and control in a distributed problem solver," *IEEE Trans. Computers*, vol. C-29, no. 12, pp. 1104–1113, December 1980.

[34] K. Hwang, W. J. Croft, G. H. Goble, B. W. Wah, F. A. Briggs, W. R. Simmons, and C. L. Coates, "A unix-based local computer network with load balancing," *IEEE Computer*, vol. 15, no. 4, pp. 55–64, April 1982.

[35] L. M. Ni, C. W. Xu, and T. B. Gendreau., "A distributed drafting algorithm for load balancing," *IEEE Trans. Software Engr.*, vol. SE-11, no. 10, pp. 1153–1161, October 1985.

[36] K. G. Shin and Y.-C. Chang, "Coordinated load sharing in hypercube multicomputers," Submitted for publication, 1990.

[37] K. G. Shin, C. M. Krishna, and Y.-H. Lee, "Optimal resource control in periodic real–time environments," *Proc. of 9th Real–time System Symposium*, pp. 33–41, 1988.

[38] M. H. Woodbury and K. G. Shin, "Evaluation of the probablity of dynamic failure and processor utilization for real–time systems," *Proc. of 9th Real–time System Symposium*, pp. 222–231, 1988.

[39] Y.-T. Wang and R. J. T. Morris, "Load sharing in distributed systems," *IEEE Trans. Comput.*, vol. C-34, no. 3, pp. 204–217, March 1985.

[40] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Trans. Software Engr.*, vol. SE-12, no. 5, pp. 662–675, May 1986.

[41] J. F. Kurose and S. Singh, "A distributed algorithm for optimum static load balancing in distributed computer systems," *Proc. of 6th IEEE Real-Time Systems Symposium*, pp. 458–467, 1985.

[42] P. S. Yu, S. Balsamo, and Y.-H. Lee, "Dynamic transaction routing in distributed database systems," *IEEE Trans. Software Engr.*, vol. SE-14, no. 9, pp. 1307–1318, September 1988.

[43] P. Krueger and R. Finkel, "An adaptive load balancing algorithm for a multicomputer," *Computer Science Technical Report # 539, University of Wisconsin–Madison*, 1987.

[44] A. Kratzer and D. Hammerstorm, "A study of load levelling," *Proc. of 1st IEEE Real-Time Systems Symposium*, pp. 647–652, 1980.

[45] R. Hagmann, "Process sever: sharing processing power in a workstation environment," *Proc. of 6th IEEE Int'l. Conf. on Dist. Comp. systems*, pp. 260–267, 1986.

[46] Y.-C. Chow and W. H. Kohler, "Models for dynamic load balancing in a heterogeneous multiple processor system," *IEEE Trans. Comput.*, vol. C-28, no. 5, pp. 354–361, May 1979.

[47] C.-Y. H. Hsu and J. W.-S. Liu, "Dynamic load balancing algorithms in homogeneous distributed systems," *Proc. of 6th IEEE Int'l. Conf. on Dist. Comp. systems*, pp. 216–223, 1986.

[48] A. N. Tantawi and D. Towsley, "Optimal static load balancing in distributed computer systems," *Journal of ACM*, pp. 445–465, April 1985.

[49] L. M. Ni and K. Hwang, "Optimal load balancing in a multiple processor system with many job systems," *IEEE Trans. Software Engr.*, vol. SE-11, no. 5, pp. 491–496, May 1985.

[50] L. M. Ni and K. Hwang, "Optimal load balancing for a processor system," *Proc. Int. Conf. Parallel Processing*, pp. 352–357, 1981.

[51] A. Hać and X. Jin, "Dynamic load balancing in a distributed system using a decentralized algorithm," *Proc. of 7th IEEE Int'l. Conf. on Dist. Comp. systems*, pp. 170–177, 1987.

[52] F. Douglis and J. Ousterhout, "Process migration in the sprite operating system," *Proc. of 7th IEEE Int'l. Conf. on Dist. Comput. Syst.*, pp. 18–25, 1987.

[53] A. Thomasian, "A performance study of dynamic load balancing in distributed systems," *Proc. of 7th IEEE Int'l. Conf. on Dist. Comp. systems*, pp. 178–184, 1987.

[54] T. L. Casavant, "Analysis of three dynamic distributed load–balancing strategies with varying global information requirements," *Proc. 7th IEEE Int'l. Conf. on Dist. Comp. systems*, pp. 185–192, 1987.

[55] A. Barak and A. Shiloh, "A distributed load-balancing policy for a multicomputer," *Software–Practice and Experience*, vol. 15, no. 9, pp. 901–913, September 1985.

[56] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. Software Engr.*, vol. SE-3, no. 1, pp. 85–93, January 1977.

[57] H. S. Stone, "Critical load factors in two–processor distributed systems," *IEEE Trans. Software Engr.*, vol. SE-4, no. 3, pp. 254–258, May 1978.

[58] L. Kleinrock, *Queueing Systems Vol. I: Theory.*, New York: Wiley, 1975.

[59] Y.-C. Chang and K. G. Shin, "Load sharing for hypercube multicomputers in the presence of node failures," *Proc. of* 6^{th} *Int'l Distributed Memory Computing Conference*, pp. 800–1474, April 1991.

[60] M. C. III Pease, *Methods of matrix algebra*, New York and London, 1965.

[61] L. N. Bhuyan and D. P. Agrawal, "Generalized hypercube and hyperbus structures for a computer network," *IEEE Trans. on Comput.*, vol. C-33, no. 4, pp. 323–333, April 1984.

[62] C. L. Seitz, "The Cosmic cube," *Commun. of the Assoc. Comp. Mach.*, vol. 28, no. 1, pp. 22–33, 1985.

[63] B. Becker and H. U. Simon, "How robust is the n-cube ?," *Proc. 27-th Annual Symposium on Fundations of Computer Science*, pp. 283–291, 1986.

[64] J. M. Gordon and Q. F. Stout, "Hypercube message rounting in the presence of faults," *Proc. of* 3^{th} *Conf. on Hypercube Concurrent Computers and Applications*, pp. 318–327, March 1988.

[65] K. W. Ryu and J. JaJa, "Efficient algorithms for list ranking and for solving graph problems on the hypercube," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, no. 1, pp. 83–119, January 1990.

[66] M.-S. Chen and K. G. Shin, "Adaptive fault–tolerant routing in hypercube multicomputers," *IEEE Trans. Comput.*, vol. C–39, no. 12, pp. 1406–1416, December 1990.

[67] T. C. Lee and J. P. Hayes, "Routing and broadcasting in faulty hypercube computers," *Third Conf. on Hypercube Concurrent Computers and Applications*, pp. 62–66, January 1988.

[68] P. Ramanathan and K. G. Shin, "Reliable broadcast in hypercube multicomputers," *IEEE Trans. Comput.*, vol. C–37, no. 12, pp. 1654–1657, December 1988.

[69] M.-S. Chen and K. G. Shin, "Depth–first search approach for fault–tolerant routing in hypercube multicomputers," *IEEE Trans. Parallel and Distributed Systems.*, vol. 1, no. 2, pp. 152–159, February 1990.

[70] M.-S. Chen and K. G. Shin, "Processor allocation in an n-cube multiprocessor using gray codes," *IEEE Trans. on Comput.*, vol. C–36, no. 12, pp. 1396–1407, December 1987.

[71] R. M. Bryant and R. A. Finkel, "A stable distributed scheduling algorithm," *Proc. of 2th IEEE Int'l. Conf. on Dist. Comp. systems*, pp. 314–323, 1981.

[72] F. C. H. Lin and R. M. Keller, "Gradient model: a demand-driven load balancing scheme," *Proc. of 6th IEEE Int'l. Conf. on Dist. Comput. Syst.*, pp. 329–336, 1986.

[73] F. Harary, *Graph Theory*, Addison-Wesley, 1969.

[74] W. Zhao and J. A. Stankovic, "Performance analysis of fcfs and improved fcfs scheduling algorithms for dynamic real–time computer systems," *Proc. of 10th Real–time System Symposium*, pp. 156–165, 1989.

[75] K. G. Shin and C. Hou, "Design and evaluation of effective load sharing in distributed real-time systems," *Proc. (in press) Third IEEE Symp. on Parallel and Distributed Processing*, December 1991.