ACCEPTED
FACULTY OF GRADUATE STUDIES
DEAN
DATE 91/08/21

FAULT-TOLERANT DISTRIBUTED REAL-TIME SCHEDULING

by

Anand Srinivasan
BSc (Hons), University of Delhi, New Delhi, 1986
MCA, Jawaharlal Nehru University, New Delhi, 1989

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the Department of
Computer Science

We accept this thesis as conforming
to the required standard

Dr. G. C. Shoja, Supervisor, Dept. of Computer Science.

Dr. D. Hoffman, Departmental Member, Dept. of Computer Science

Dr. F. El Guibaly, Outside Member, Dept. of Elect. & Comp. Eng.

Dr. K. F. Li, External Examiner, Dept. of Elect. & Comp. Eng.

© Anand Srinivasan, 1991
UNIVERSITY OF VICTORIA

*All rights reserved. This thesis may not be reproduced
in whole or in part by photocopy or other means,
without the permission of the author.*

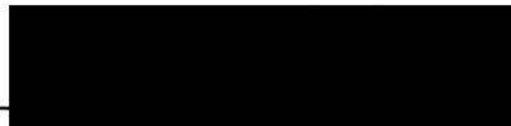
QA76.54
S65

Supervisor: Dr. G. C. Shoja

ABSTRACT

Hard Real-Time Systems are employed in highly complex and time-critical applications where a high degree of fault-tolerance is a mandatory feature. A distributed algorithm for providing fault-tolerant optimal schedule in a simply periodic distributed real-time system is proposed. Each job is serviced by a primary or an alternate algorithm. The primary algorithm provides a desirable service that may not satisfy the timing constraints, whereas the alternate algorithm provides an acceptable service that always satisfies the timing constraints. After invoking an optimal scheduler in each individual node, the algorithm tries to schedule additional primaries on other nodes. Since primaries provide more accurate results, the distributed scheduling algorithm achieves better result accuracy without sacrificing the timing accuracy. The algorithm is first introduced and then applied to virtual ring network and binary n-cube interconnection network. A dynamic scheduler which enhances the run-time performance of the previously scheduled jobs, is also described. The results of performance tests for various randomly generated data are also given.

Examiners:



Dr. G. C. Shoja, Supervisor, Dept. of Computer Science

Dr. D. Hoffman, Departmental Member, Dept. of Computer Science

Dr. F. El Guibaly, Outside Member, Dept. of Elect. & Comp. Eng

Dr. K. F. Li, External Examiner, Dept. of Elect. & Comp. Eng

Table of Contents

Title Page i

Abstract ii

Table of Contents iv

List of Figures vii

Acknowledgement x

Dedication xi

1 Introduction 1

1.1 Real-Time Systems 1

1.1.1 Features 1

1.1.2 Constraints 2

1.2 Real-Time Scheduling 3

1.2.1 Static Scheduling Algorithms 3

1.2.2 Dynamic Scheduling Algorithms 5

1.3 Fault-Tolerance for Real-Time Systems 7

1.3.1 Architecture and Hardware 7

1.3.2 Software Fault-Tolerance 8

1.4 Fault-Tolerant Scheduling 9

1.5 Objectives of the Thesis 10
1.6 Structure of the Thesis 11

2 Software Fault-Tolerance Schemes 12
2.1 Introduction 12
2.2 N-Version Programming Scheme 13
2.3 Recovery Block Scheme 16
2.4 The Deadline Mechanism 19

3 Proposed Methods and Solutions 22
3.1 Introduction 22
3.1.1 Definitions and Notations 22
3.2 Distributed Fault-Tolerant Scheduler 24
3.2.1 Our Proposed Alternative Scheduler 25
3.2.2 Our Proposed Distributed Scheduler 27
3.2.3 Dynamic Scheduling 31

4 System Configuration 37
4.1 Virtual Ring 37
4.1.1 Initial Schedule 37
4.1.2 Collection of Unscheduled Primaries 38
4.1.3 Communication in Virtual Ring 38
4.1.4 Dynamic Scheduling in Virtual Ring 41
4.2 Binary n-cube 46
4.2.1 Initial Schedule 46
4.2.2 Collection of Unscheduled Primaries 47
4.2.3 Communication in Binary n-Cube 49
4.2.4 Dynamic Scheduling in Binary n-Cube 55

5 Performance Evaluation **60**

5.1 Introduction 60

5.2 Generation of Test Cases 60

5.3 Results of Simulation for Virtual Ring 61

5.3.1 Performance for Two Levels in Virtual Ring 61

5.3.2 Performance for Three Levels in Virtual Ring 62

5.3.3 Performance for Higher Levels in Virtual Ring 62

5.4 Results of Simulation for Binary n-Cube 77

5.4.1 Performance for Two Levels in Binary n-Cube 77

5.4.2 Performance for Three Levels in Binary n-Cube 78

5.4.3 Performance for Higher Levels in Binary n-Cube 79

6 Conclusions **96**

Bibliography **97**

List of Figures

3.1 Example for a simply periodic system 23
3.2 The fields of the linked list element 28

4.1 Execution time for an exact simply periodic system of 3 nodes 38
4.2 Schedule in individual nodes after Phase 1 39
4.3 Linked list of unscheduled primaries 40
4.4 Linked list after communication with other nodes 41
4.5 New schedule indicating some primaries of Node 0 scheduled at Node 2 42
4.6 Linked list of node 1 after Phase 3 Algorithm 43
4.7 Linked list constructed using the initial table 43
4.8 Global table constructed by node 1 44
4.9 Group of sorted job lists for node 1 44
4.10 The updated schedule of node 1 45
4.11 The updated working table for Node 1 46
4.12 Execution time for various primary and alternate algorithms for nodes in 3-cube. 47
4.13 Scheduling individual nodes using Phase 1 48
4.14 Linked list formed after Phase 2 50
4.15 Hamiltonian path in 3-cube and corresponding Gray code. . . 52
4.16 Empty slots before and after Phase 3 algorithm. 53

4.17 New schedule after Phase 3 algorithm. 54
4.18 Linked List of node 001 after Phase 3 Algorithm 55
4.19 Linked list constructed using the initial table 56
4.20 Global table constructed by node 001 56
4.21 Group of sorted job lists for node 001 57
4.22 The updated schedule of node 001 58
4.23 The updated working table of Node 001 59

5.1 Performance of the scheduler for virtual ring of 3 nodes 64
5.2 Performance of the scheduler for virtual ring of 4 nodes 65
5.3 Performance of the scheduler for virtual ring of 5 nodes 66
5.4 Performance of the scheduler for virtual ring of 6 nodes 67
5.5 Percentage of test cases where extra primaries were scheduled for virtual ring network 68
5.6 Performance of the scheduler for virtual ring of 3 nodes 69
5.7 Performance of the scheduler for virtual ring of 4 nodes 70
5.8 Performance of the scheduler for virtual ring of 5 nodes 71
5.9 Performance of the scheduler for virtual ring of 6 nodes 72
5.10 Percentage of test cases where extra primaries were scheduled for virtual ring network 73
5.11 Percentage of test cases where extra primaries were scheduled for virtual ring network 74
5.12 Percentage of test cases where extra primaries were scheduled for virtual ring network 75
5.13 Average number of extra primaries scheduled when $T_{k+1}/T_k = 2$ 76
5.14 Performance of the scheduler for Binary 2-Cube 80
5.15 Performance of the scheduler for Binary 3-Cube 81

5.16 Performance of the scheduler for Binary 4-Cube 82
5.17 Percentage of test cases extra primaries were scheduled for n-Cube 83
5.18 Percentage of test cases extra primaries were scheduled for n-Cube 84
5.19 Performance of the scheduler for Binary 2-Cube 85
5.20 Performance of the scheduler for Binary 3-Cube 86
5.21 Performance of the scheduler for Binary 4-Cube 87
5.22 Percentage of test cases extra primaries were scheduled for n-Cube 88
5.23 Percentage of test cases extra primaries were scheduled for n-Cube 89
5.24 Percentage of test cases extra primaries were scheduled for n-Cube 90
5.25 Percentage of test cases extra primaries were scheduled for n-Cube 91
5.26 Percentage of test cases extra primaries were scheduled for n-Cube 92
5.27 Percentage of test cases extra primaries were scheduled for n-Cube 93
5.28 Average number of extra primaries scheduled for n-Cube . . . 94
5.29 Average number of extra primaries scheduled for n-Cube . . . 95

Acknowledgment

I would like to thank my supervisor, Dr. G. C. Shoja of the Department of Computer Science, for his encouragement, patience, and advice during the course of this research and during the preparation of this manuscript. I would also like to thank Prof. F. Ruskey and Dr. I. Sharf for their help.

Financial assistance, received in the form of a fellowship from the University of Victoria, is gratefully acknowledged.

To my parents

Chapter 1

Introduction

1.1 Real-Time Systems

1.1.1 Features

Real-time systems differ from traditional computer systems in that deadlines or other explicit timing constraints are attached to tasks, the systems are in a position to make compromises, and faults, including timing faults, may cause catastrophic consequences. Real-time systems are characterized by the fact that severe consequences will result if logical as well as timing correctness properties of the system are not satisfied. This implies that, unlike many systems where there is a separation between correctness and performance, in real-time systems correctness and performance are very tightly interrelated.

Real-time computing is a wide open research area of challenging problems. The current state of art in real-time computing is elaborated by Stankovic in [1]. Real-time systems are used in time-critical applications, such as command and control systems [2] , nuclear power plants, process and flight control systems [3] , space shuttle and aircraft avionics [4], and robotics [5].

Future generation real-time systems will be used in similar application

areas as the current ones, but will be more complex in that they will be distributed, contain highly dynamic and adaptive behavior, exhibit intelligence, and will have long lifetimes. Examples of these more sophisticated systems are the autonomous land rover, controllers of robots with elastic joints, systems found in intelligent manufacturing, the space station, and undersea exploration.

1.1.2 Constraints

As indicated earlier, in real-time applications the correctness of computation depends not only on the results of computation but also the time at which outputs are generated. Since real-time systems mostly handle time critical applications, the execution of various tasks has to be completed before their respective deadlines.

Wirth [6] has classified programs into three types : sequential, parallel and processing-time dependent (real-time). The difficulty of specification, design and analysis of programs increases as parallelism is incorporated and increases further when real-time constraints are introduced. The problem of specification, design and analysis of real-time systems with timing constraints have been actively researched [7,8].

In addition to the timing constraints, a task may also possess the following types of constraints and requirements :

Resource Constraints : A task may require access to certain resources other than the CPU such as I/O devices, data structures, files and databases.

Precedence Relationships : A complex task, for example, one requiring access to many resources, is better handled by breaking it up into multiple subtasks related by precedence constraints and each requiring a subset of the

resources.

Concurrency Constraints : Tasks should be allowed concurrent access to resources provided the consistency of the resources is not violated.

Communication Requirements : Sets of cooperating tasks will be the norm for distributed, hard real-time systems. The semantics of the communications will vary as well as the interconnection structure between the communicating tasks and their timing requirements.

Placement Constraints : When multiple instances of a task are executed for fault-tolerance, the different instances should be executed on different processors.

Criticalness : Depending on the functionality of a task, meeting the deadline of one task may be considered more critical than another.

1.2 Real-Time Scheduling

In a real-time system, task scheduling is an important problem, because it is the scheduling algorithm that ensures that tasks meet their deadlines. Task scheduling in real-time systems can be either *static* or *dynamic*. A *static* approach calculates schedules for tasks off-line, and therefore requires complete prior knowledge of tasks' characteristics. A *dynamic* approach determines schedules for tasks at run-time and allows tasks to be dynamically invoked.

1.2.1 Static Scheduling Algorithms

1.2.1.1 Static Scheduling in Centralized Systems

Horn [9] has developed two simple preemptive scheduling algorithms for real-time tasks in uniprocessor systems. The first algorithm, with time complexity

of $O(n^2)$, schedules tasks with arbitrary ready times and deadlines, where n is the number of tasks to be scheduled. The other algorithm, with time complexity of $O(n^3)$, schedules tasks with arbitrary ready times and deadlines for multiprocessor systems.

Efficient scheduling algorithms have been developed for periodic tasks. For uniprocessor systems, Liu and Layland [10] developed a scheduler which assigns higher priorities to tasks with shorter periods. Lehoczky and Sha [11] describe a technique to modify the periods of tasks in such a way that, while tasks' timing constraints continue to be met, better processor utilization is achieved. Scheduling periodic tasks on multiprocessor systems is more complicated. The set of periodic tasks are partitioned among a minimum number of processors such that each partition of the periodic tasks can be scheduled on one processor according to earliest deadline scheme. Bannister and Trivedi [12] have proposed a simple best-fit partition scheme. Various preemptive schedulers [13,14] and nonpreemptive schedulers [15–19] have been researched.

Lawler [20] has presented a solution to the problem of scheduling nonpreemptive tasks with precedence constraints for uniprocessor systems. Blazewicz [21] has proved that, for this scheduling problem, a preemptive schedule exists if and only if a nonpreemptive schedule exists. Kasahara and Narita [5] have developed a heuristic search algorithm to determine the minimum schedule length for a set of non-preemptive tasks with arbitrary precedence constraints. Manacher [22] and Ullman [16,23] have studied the problem of scheduling tasks with precedence constraints in multiprocessor systems.

1.2.1.2 Static Scheduling in Distributed Systems

Considerable effort has been spent on developing heuristic algorithms to find suboptimal solutions to the problem of scheduling real-time tasks in distributed systems [24]. Stone [25,26] and Lo [27] have developed network flow algorithms to allocate tasks with arbitrary communication patterns in dual-processor and three-processor systems. Efe [28] developed a heuristic allocation algorithm to balance processor load and to minimize communication cost. Ma *et al.* [29] developed an integer programming model to find optimal allocation for tasks with explicit timing constraints. Leinbaugh and Yamini [30] developed an analysis algorithm to compute the worst case finish time for a set of tasks which run on a dedicated network. Peng and Shin [31] developed a generalized stochastic Petri net to model the behavior of a distributed hard real-time system.

1.2.2 Dynamic Scheduling Algorithms

1.2.2.1 Dynamic Scheduling in Centralized Systems

Most of the algorithms which are optimal for static scheduling are not optimal for dynamic scheduling. Mok and Dertouzos [32] showed that, for multiprocessor systems, there can be no optimal algorithm for scheduling preemptable tasks if the arrival time of tasks are not known *a priori*.

For uniprocessor systems, Dertouzos [33] showed that the earliest deadline algorithm is optimal for scheduling preemptable tasks with arbitrary arrival times. Ramamritham and Stankovic [34] have described a guarantee scheme based on the earliest deadline policy, which also takes run-time costs into account. For scheduling nonpreemptable tasks, Baker and Su [24] com-

pare four simple heuristic algorithms which schedule tasks according to an order determined by ready time, deadline, the average of the ready time and deadline, and by both the ready time and the deadline, respectively.

For scheduling preemptable tasks on multiprocessor systems, Mok and Dertouzos [35] showed that, if a set of tasks can be scheduled assuming their start times are same, then this same set of tasks can also be scheduled at run-time if their start times are different. Locke, Tokuda and Jensen [36] have compared a number of simple dynamic scheduling policies. Zhao, Ramamritham, and Stankovic [37,38] have developed a heuristic function and an efficient backtracking scheme for scheduling preemptable and nonpreemptable tasks with resource constraints.

1.2.2.2 Dynamic Scheduling in Distributed Systems

A dynamic scheduling algorithm for a distributed system should maximize the guarantee ratio of tasks, i.e., the total number of tasks guaranteed versus the total number of tasks that arrive in the network-wide system. Many distributed scheduling algorithms have been proposed for traditional distributed systems. The objective of these algorithms has been to balance loads among nodes in a system and are therefore referred to as *load-balancing* algorithms in the literature [39–45]. Ramamritham and Stankovic [34] have developed a heuristic algorithm for scheduling mutually independent tasks in distributed real-time systems. Their goal is to schedule tasks such that as many tasks as possible can be guaranteed to complete before their deadlines. Simulation results of this algorithm are reported in [46]. These results show that dynamic and distributed real-time scheduling is feasible and that a system can benefit substantially from distributed scheduling under a wide range of

system conditions and task parameters.

Precedence constraints of tasks add one additional dimension to the complexity of dynamic scheduling problem in distributed systems. Cheng *et al.* [47] developed a novel algorithm to solve this problem when each group has a deadline. For a group that must be distributed, their approach attempts to partition tasks in the group into subgroups and distribute the subgroups in the network to be scheduled in parallel.

1.3 Fault-Tolerance for Real-Time Systems

1.3.1 Architecture and Hardware

The increasing application of computers to real-time control functions has created situations in which a computer failure could result in unacceptably high costs, either in terms of life or property. Such systems therefore, require a high degree of fault-tolerance.

A critical design issue for any fault-tolerant system is redundancy management, i.e., the control of resources for fault-tolerance. Fault-tolerance in a real-time system is based on masking errors, either for the duration of the mission (static redundancy), or until the error can be isolated and the system reconfigured (dynamic redundancy). Static redundancy requires massive physical redundancy, but needs only simple redundancy management. Dynamic redundancy minimizes extra hardware and allows rescheduling and reallocation of tasks. It also provides graceful reconfiguration, by excluding nodes as they fail and readmitting them as they are repaired. Dynamic redundancy management depends on system consistency, even in the presence of faults.

A landmark development in the search for extreme reliability in real-time

control systems was the Software Implemented Fault-Tolerance (SIFT) computer system [3,48,49,50].. The SIFT project pioneered both theoretically provable fault-tolerance and system consistency. SIFT has a large system overhead since the fault-tolerance and executive functions are implemented in software on the same processor which performs the application tasks [51]. Scheduling for all tasks in SIFT is static [51,52].

A contemporary alternative to SIFT was the Fault-Tolerant Multiprocessor (FTMP) architecture [53]. FTMP explored hardware triple modular redundancy (TMR), priority based scheduling, exact voting and tight synchronization.

A Multicomputer Architecture for Fault-Tolerance (MAFT) exploits the various fault-tolerant methodology [54]. The architectural approach taken in MAFT provides extreme reliability without sacrificing performance, flexibility, or programmability. Through the separation of system overhead from application functions, MAFT improves resource availability to application tasks.

1.3.2 Software Fault-Tolerance

Real-time applications require continuity of correctly computed output, and that implies correct performance of both hardware and software. Fault-tolerance is achieved by incorporating redundancy in the system. Redundancy in fault-tolerant software requires programs that are deliberately different from the original ones which they are intended to backup. The Recovery Block Scheme pioneered by Horning *et al.* [55] and Randell [56] meets these requirements. Hecht [57] developed a technique for reliability analysis of a software system illustrating the application of the recovery block to

real-time programs.

Other well known software fault-tolerance techniques are the N-Version Programming [58], and the Deadline Mechanism [59]. The Deadline mechanism, is a variation of the Recovery Block Scheme. These techniques will be described in more detail in Chapter 2.

1.4 Fault-Tolerant Scheduling

When redundancy is introduced in the software system to achieve fault-tolerance, the number of tasks to be executed increases dramatically. An efficient scheduler is needed so that tasks can be scheduled in such a way that usage of additional resources is minimized. A fault-tolerant scheduler achieves a good balance between satisfying the timing constraints and achieving result accuracy.

An event driven, priority based, globally verifiable scheduling mechanism has been devised for MAFT. The MAFT scheduler follows the framework for fault-tolerant real-time software proposed by Anderson and Knight [60]. For a given system configuration, task-to-node allocations are static. The task selections in MAFT are based upon periodicity requirements, precedence relationships, and task priorities.

In a hard real-time system, a timing fault is said to occur when a real-time process delivers its result too late or too early. To avoid such timing faults, *imprecise computation approach* has been introduced [61–64]. Instead of incorporating redundancy into system to achieve acceptable result, this approach relies on making available results that are of poorer, but acceptable quality on a timely basis when results of the desired quality cannot be produced. The problem of scheduling periodic jobs in hard real-time systems

that support imprecise computations is discussed by Chung *et al.* [61].

Multiple versions of the tasks called *clones* were used by Krishna *et al.* [65] to achieve fault-tolerance. There are two types of clones namely *primary* and *ghost*. A primary clone is executed normally, whereas the ghost clone is a backup copy which lies dormant until it is activated to take the place of a corresponding primary. A fault-tolerant scheduler based on this method was introduced to achieve quick recovery from failure.

Campbell *et al.* [59] and Wei *et al.* [66], proposed and applied a method based on the Recovery Block scheme, namely the Deadline Mechanism. The Deadline Mechanism replaces the acceptance test of the Recovery Block by a centralized scheduler and supervisor. The Deadline Mechanism requires each job to have a *primary* algorithm, which provides a service that is more accurate but may not satisfy the timing constraints, and an *alternate* algorithm, which is less accurate but does satisfy the timing constraints. Liestman and Campbell implemented the mechanism for a scheduling problem in a simply periodic system [67].

1.5 Objectives of the Thesis

We propose an algorithm that can schedule additional primaries on nodes of a distributed system, where the Deadline Mechanism has already been employed to schedule the jobs on individual nodes.

Our goal is to schedule the maximum number of primary algorithms possible in the distributed system. When a primary algorithm scheduled using the static scheduler is executed successfully, the alternate algorithm need not be executed, even if already scheduled. However, the knowledge that a primary algorithm satisfying the timing constraint can be executed successfully

before its deadline is available only during the run-time. Thus our other objective is to find an efficient run-time scheduler which can dynamically schedule unscheduled primary algorithms in the time slots occupied by the redundant alternate algorithms. Chapter 3 discusses the dynamic scheduler in detail.

1.6 Structure of the Thesis

The remainder of the thesis is organized as follows. Chapter 2 gives background on software fault-tolerance techniques. Chapter 3 describes our proposed fault-tolerant scheduling algorithm for a distributed real-time system. Chapter 4 extends the scheduler to distributed real-time systems with various configurations such as virtual ring and n-cube interconnection networks. Chapter 5 presents the results achieved when the scheduler was simulated, and gives the performance evaluation of the algorithm. Chapter 6 concludes the thesis.

Chapter 2

Software Fault-Tolerance Schemes

2.1 Introduction

Techniques for dealing with hardware faults in computers for critical applications have been under investigation for considerable time [68,69]. Lately, much research effort has been directed towards achieving software fault-tolerance [57].

Software fault-tolerance techniques use redundancy in software as the main tool to achieve reliability. Usually, multiple copies of an algorithm are executed and one of the results is chosen. An efficient scheduler is also necessary in order to strike a balance between meeting deadlines and achieving result accuracy.

In this chapter, two important software fault-tolerance techniques, namely N-version programming and the Recovery Block Scheme are described. This is followed by a brief description of a fault-tolerant technique for real-time systems namely the Deadline Mechanism.

2.2 N-Version Programming Scheme

One way to provide software fault-tolerance is by introducing redundancy in the system. N-version programming is a prominent software fault-tolerance technique introduced by Chen and Avizienis [58].

N-version programming, as the name suggests, has N versions of a program ($N \geq 2$) which have been independently designed to satisfy a common specification. The N versions are executed and their results compared by some form of replication check, based on a majority vote. This check can eliminate erroneous results and pass on the presumably correct results calculated by the majority to the rest of the system.

Control of the N versions of a program is provided by what is termed the driver program. The driver program is responsible for :
(i) invoking each of the versions;
(ii) waiting for the versions to complete their execution;
(iii) comparing and acting upon the N sets of results.

It is obvious that mechanisms are required to synchronize the actions of the driver and the versions, and to communicate outputs from the versions to the driver. The scheme also requires that each version be executed atomically and have access to the same input space.

A synchronization mechanism is an important aspect of N-Version programming. The scheme proposed by Chen and Avizienis is fairly simple and is based essentially on the use of **wait** and **send** primitives. The versions wait and do not commence processing until a **send** is executed by the driver. Similarly, the driver **waits** until **send** responses have been received from all N versions to indicate that their outputs are complete. The voting check on

the sets of results can then be evaluated. The synchronization scheme has to allow for different execution times of the modules, in particular for modules which do not complete their execution, for example due to an infinite loop caused by a design fault. This requires some form of timeout mechanism to be added to the synchronization mechanism used by the driver. A different approach to synchronization is adopted in the SIFT and Space Shuttle systems, namely, an N Modular Redundancy structure for executing programs. The problem of versions which do not complete is overcome in these systems by performing the voting check at a predetermined fixed time rather than waiting for completion signals from the versions. As a result, both systems contain a complex synchronization scheme to ensure that the executions of the versions on multiple processors do not get out of step. The synchronization in these systems is effected by the driver whereas in the N-Version programming scheme the synchronization is driven by the versions.

Multiprocessor systems would be suitable for executing such programs since it could then be arranged to execute each version in parallel on independent hardware. Execution of an N-version program on a single processor system is possible so long as the isolation of separate versions can be maintained. When executed, each version must have access to an identical set of input values. One method of implementing this would be for the driver to communicate the set of input values to each version, although there is the danger of there being a large set of input values. Another possibility would be to allow the versions to access the input values from a shared, global data structure.

Probably the most important aspect of the N-version programming scheme is the voting check performed by the driver program. For applications where versions can be expected to produce identical results if their execution is

without fault, an equality voting check can be used with the majority being selected as the correct result. However, for some computations an exact comparison check cannot be used, since inexact hardware representation of real numbers coupled with the different algorithms used can lead to minor discrepancies between valid sets of results. For this reason, *Inexact Voting* has been investigated [70], where a check can identify a consensus even though small discrepancies occur between the sets of results being compared. The obvious approach is to use some form of range check, expecting that the results will be within a certain range of each other. As noted by Chen and Avizienis [58], the maximum allowable range used in a check may be difficult to determine and may change between executions of a version. Even if the range can be determined and all the versions produce results, the inexact vote requires a non-trivial algorithm to identify the erroneous values in a set and then evaluate a result derived from the remaining values.

When a voting check can be successfully implemented, N-version programming is a simple and attractive framework for fault tolerance. Error detection is provided by the voting check; damage assessment is not required if the activities of the versions are atomic. Error recovery involves ignoring the values identified as erroneous by the check; and fault treatment simply results, in the versions determined to have produced erroneous results being ignored. Of course, if a majority of versions do not produce equivalent results then further fault tolerance measures will be needed to avert failure. Two-Version systems [71] are one instance of this situation since a voting check, while satisfactory if the two sets of results are in agreement, can only indicate detection of an error when the two sets differ. The Two-Version system relies on manual intervention when the check detects an error. If fault-tolerance in a Two-Version system had to be provided without manual

assistance, some form of acceptance test would be necessary to differentiate between the acceptable and non-acceptable sets of results.

2.3 Recovery Block Scheme

The Recovery Block Scheme for providing software fault-tolerance in sequencial programs was introduced by Horning *et al.* [55] and was later extended by Randell [56].

Consider a task that has to be performed reliably by a software system, and assume that a non-redundant software module has been designed and implemented with the aim of satisfying the specification of this task. This module will be referred to as the *primary module.* It is assumed that this module has been tested and debugged as much as was practicable. However, it is recognized that the module may still contain residual design faults which could lead to system failures.

The reasonableness of the results calculated using the *primary module* is checked using an *acceptance test.* The *acceptance test* will consist of a sequence of statements which will raise an exception if the state of the system is not acceptable.

If the execution of the *primary module* produces an unacceptable result, then the system state is restored to the state that existed just before the *primary module* was entered. The same procedure is continued with several *alternate modules* until an acceptable result is obtained. If the primary module and all the alternate modules end up providing unacceptable results then an error routine is invoked.

Thus a normal Recovery Block would look like :

```
Establish Recovery Point;
ensure  < acceptance test >
by      < Primary Module >
else by < Alternate Module #1 >
else by < Alternate Module #2 >
else by < Alternate Module #3 >
        .
        .
        .
else by < Alternate Module #n >
else error.
```

The acceptance test, which is common to all modules, is identified by the keyword **ensure**, and is situated at the beginning of the recovery block. Following the acceptance test is the primary module, identified by the keyword **by**, and a set of alternate modules each of which is preceded by **else by**. The final **else error** clause emphasizes the fact that no further alternate modules remain.

On initial entry, a recovery point is established and the primary module is entered. On completion of the module the acceptance test is evaluated. If this test, or the execution of the module, does not raise any exceptions then the results of the module are assumed to be acceptable and the recovery block is exited. However, if an exception is raised then automatic restoration of the recovery point occurs. Following recovery, the sequence of execution described above is repeated except that the next module is used in place of the module that failed. If all the modules fail, then this is regarded as a failure of the recovery block and an appropriate exception will be signalled.

The Recovery Blocks can be nested so that one recovery block can form part of a module of an enclosing recovery block. An exception resulting

from a failure of an inner recovery block will simply cause termination of the enclosing module in the same manner as any other exception.

The Recovery Block Scheme does not impose any constraints on the programming style, methodology and language used to implement the modules and the acceptance test [72]. Since the primary module is the first module in a recovery block to be executed, it is normal to use as the primary module a module which has characteristics that make it more desirable than those modules selected as the alternates. For example, the primary module might have the shortest execution time or use the least amount of main storage. The following recovery block illustrates a fault-tolerant sort program.

```
ensure  ( A[j+1] >= A[j] ) for j = 1,2,...n-1
by      Sort A using Quick sort
else by Sort A using Shell sort
else by Sort A using Insertion sort
else error.
```

This recovery block is intended to sort an array A into ascending order. The primary module uses what is hoped will be the most efficient sorting algorithm. Successive alternate modules also aim to sort A correctly, but using less and less efficient algorithms. Hopefully, these less efficient algorithms will be simpler than the algorithm employed by the primary module and hence will be less prone to design faults.

It is not necessarily the case that all the modules in a recovery block produce exactly the same results. The constraint on the modules is that they produce acceptable results, as defined by the acceptance test. Thus, while the primary module attempts to produce the desired results, the second and subsequent modules may only attempt to provide an increasingly degraded

service.

The Recovery Block scheme increases the overall size of a programming task because additional software must be designed and implemented for the alternate modules. The designer of one module needs to have no knowledge of the design of any of the other modules. The execution of the acceptance test, however, contributes an increase in complexity of a recovery block program when compared to a program without provision for fault-tolerance.

2.4 The Deadline Mechanism

A variation of the recovery block scheme called the *Deadline Mechanism* has been proposed by Campbell *et al.* [59]. The Deadline Mechanism is intended to support software fault-tolerance in real-time applications where a program has to satisfy requests for service with a given deadline or else system failure is likely to ensue.

The Deadline Mechanism requires each service component to have a primary and an alternate algorithm. The primary algorithm provides a service which is in some sense more desirable. The alternate algorithm meets the specifications for that service component but may be less desirable. A scheduling algorithm ensures that each service request is satisfied by either the primary or the alternate algorithm.

Scheduling primaries or alternates reliably to meet real-time constraints requires the calculation of the execution period of each algorithm. This bound may be determined by a theoretical computation from the terminating conditions of the algorithm. The accuracy of determination of the execution periods is critical to system performance and reliability.

If the primary completes within its execution period, its results are used

in preference to those of the alternate. If the primary should fail to complete within its execution period, because of a timing fault in the primary or a miscalculation of the execution period, the results from the alternate are used. If the alternate is run before the primary, a cache may be used to hold results from the alternate until the primary either fails or completes successfully.

An example of how a navigation program could be specified for the Deadline Mechanism is given below. The example illustrates the way Deadline Mechanism provides fault-tolerance.

```
every  second
within
calculate by   read sensors  (Primary Module)
calculate new position

else by approximate new position
from old position (Alternate Module)
```

In this notation, the **every** statement is used to specify the maximum frequency with which the program is invoked. The **within** statement determines the deadline that must be met by the program, by specifying the maximum amount of time that can elapse before results must be provided. Following this are the primary module, and a single alternate module used to provide a degraded service should the primary fail. It is assumed that the alternate module is free from faults.

Since the program must provide its service before the deadline elapses, it is necessary for the program designer to estimate the execution time of the

program accurately. The Deadline Mechanism requires only that a guaranteed upper bound can be placed on the time needed to execute the alternate module. The methods used for scheduling are discussed in the following section.

Chapter 3

Proposed Methods and Solutions

3.1 Introduction

As stated earlier, Liestman and Campbell have implemented the mechanism for a scheduling problem in a simply periodic system [67]. We propose a distributed algorithm that can schedule additional primaries on nodes of a distributed system, where the Deadline Mechanism has already been employed to schedule the jobs on individual nodes.

3.1.1 Definitions and Notations

Definition : The *arrival period* for a job is defined as the duration between subsequent requests.

Definition : A *simply periodic system* is one in which the arrival period for a job is of fixed length and a multiple of the next smallest arrival period.

For example, as shown in Figure 3.1 in a *simply periodic system* the arrival period for three jobs J_0, J_1 and J_2 could be 10,20 and 40 respectively,

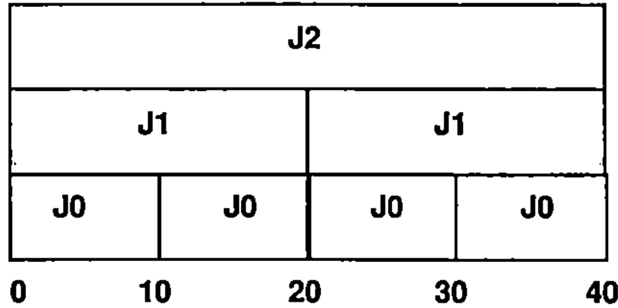


Figure 3.1: Example for a simply periodic system

where 0,1 and 2 are the *levels* of J_0, J_1 and J_2.

Definition : An *exact simply periodic distributed* system constitutes a collection of simply periodic systems with same arrival period.

Definition : A schedule is *fault-tolerant feasible* if all requests will be satisfied before their deadlines, even if none of the scheduled primary algorithms succeed.

Definition : A schedule is *fault-tolerant optimal* if it is feasible and has the maximum number of primaries scheduled among all *fault-tolerant feasible* schedules.

Notation :

N : Total number of nodes

L : Total number of levels

$P_{l,i}$: Primary in a node i for level l

T_l : Arrival Period for a job $J_{l,i}$ of a node i

$E(P_{l,i})$: Execution Time of $P_{l,i}$

$E(A_{l,i})$: Execution Time of $A_{l,i}$

3.2 Distributed Fault-Tolerant Scheduler

We consider an exact simply periodic real-time system of N nodes. We first schedule the set of jobs $\{J_{l,i} | 0 \leq l < L\}$ arriving at a simply periodic node i of the distributed system using the Liestman-Campbell Scheduler [67]. A job $J_{k,i}$ of level k, with primary $P_{k,i}$, alternate $A_{k,i}$ and arrival period T_k is serviced T_{L-1} / T_k times. For instance, the lowest *level* contains the job $J_{0,i}$ with arrival period T_0, and is serviced T_{L-1} / T_0 times. An *exact simply periodic distributed* system with a maximum level of L, has the same arrival period $T_l, 0 \leq l < L$ for all nodes.

The algorithm given below creates an optimal fault-tolerant schedule for one entire T_{L-1} period.

Given a set of jobs $\{J_{l,i} | 0 \leq l < L\}$ for which a fault-tolerant schedule is feasible, a fault-tolerant schedule exists for the period T_l which maximizes the number of primaries executed. An optimal schedule is constructed recursively as follows :

For every node i

Assume that an optimal schedule $S_{l,i}$, has been constructed for the set of jobs $J_{0,i}, J_{1,i}, \ldots, J_{l,i}$ for the period T_l and that it has the maximum amount of idle time possible. The following steps are applied :

Step 1 : Construct a provisional schedule $S_{l+1,i}$ by concatenating T_{L-1}/T_l copies of schedule $S_{l,i}$.

Step 2 : Modify $S_{l+1,i}$. Remove the minimum number of largest $P_{k,i}$'s for $0 \leq k \leq l$, increasing idle time until $A_{l+1,i}$ can be scheduled. Schedule $A_{l+1,i}$

Step 3 : If the resulting idle time is large enough, then schedule $P_{l+1,i}$.

Step 4 : Otherwise, find the largest scheduled primary $P_{k,i}$, where $0 \leq k \leq l$. If $P_{k,i} > P_{l+1,i}$, then remove $P_{k,i}$ and schedule $P_{l+1,i}$, thus maximizing idle time.

Whenever a choice is possible, lower level jobs are scheduled before higher level jobs, and within one level, jobs satisfied by both a primary and alternate are scheduled in preference to jobs satisfied by only an alternate.

The above algorithm creates an optimal fault-tolerant schedule for a set of jobs such that either a primary and an alternate or an alternate is scheduled for each request. We call this the Phase 1 of our distributed scheduling algorithm.

3.2.1 Our Proposed Alternative Scheduler

The Liestman-Campbell algorithm recursively schedules primary and alternate algorithms starting with the lowest level. The number of primaries scheduled changes continuously throughout the construction of the schedule which makes the formation of a linked list very time consuming. For example, if $S_{l,i}$ has been constructed for some $0 \leq l < L$, then in $S_{l+1,i}$ some of the primaries of $S_{l,i}$ could be taken off to schedule the alternate $A_{l+1,i}$ and $P_{l+1,i}$. Thus there is a continuous change in the number of primaries at various levels during recursion in order to maximize empty slots and accommodate the alternates. Keeping track of these changes is an overhead which we try to avoid in our proposed algorithm.

We therefore present a different scheduling algorithm which schedules only the alternates during the levels of recursion. After obtaining a schedule

for the alternates, maximum number of primaries are scheduled in the empty slots available.

For every node i.

Step 1 : Assume that we have constructed an optimal alternate schedule $S_{l,i}$, where $0 \leq l < L$ with $\{A_{k,i} | 0 < k \leq l\}$ for the time period T_l. Construct a provisional schedule $S_{l+1,i}$ by concatenating T_{L-1}/T_l copies of $S_{l,i}$. Schedule $A_{l+1,i}$ in the empty slots available.

Step 2 : For $0 \leq l < L$, schedule primary $P_{l,i}$, in the empty slots available in $[T_l$ * k, T_l * (k+1)] for $0 \leq k \leq T_{L-1}/T_l$. If $P_{l,i}$ is unschedulable, and a primary with execution time greater than $P_{l,i}$ exists in the time interval, then $P_{l,i}$ is scheduled in the place of the primary with largest execution time by maximizing empty slots.

Theorem:

A necessary and sufficient condition to have a fault-tolerant feasible schedule for alternates in a node i is, $\sum_{l=0}^{L-1}(E(A_{l,i})/T_l) < 1$ where $0 \leq i < N$

Proof :

(a) Necessary Condition

Assume we have a fault-tolerant feasible schedule. All alternates are scheduled, thus :

$(T_{L-1}/T_0)E(A_{0,i}) + (T_{L-1}/T_1)E(A_{1,i}) + ... + (T_{L-1}/T_{L-1})E(A_{L-1,i}) < T_{L-1}$

$\sum_{l=0}^{L-1}(T_{L-1}/T_l)E(A_{l,i}) < T_{L-1}$

$\sum_{l=0}^{L-1} E(A_{l,i})/T_l < 1$

(b) Sufficient Condition

Let $\sum_{l=0}^{L-1} E(A_{l,i})/T_l < 1$

then $E(A_{l,i}) < T_l$, $0 \leq l < L$

We claim that all the alternates have been scheduled. Supposing it is not, then for some level r, $0 \leq r \leq L-1$

$\sum_{l=0}^{r}(T_r/T_l)E(A_{l,i}) > T_r$

$\sum_{l=0}^{r} E(A_{l,i})/T_l > 1$

$\sum_{l=0}^{L-1} E(A_{l,i})/T_l = \sum_{l=0}^{r} E(A_{l,i})/T_l + \sum_{l=r+1}^{L-1} E(A_{l,i})/T_l > 1$

which is a contradiction since we have assumed $\sum_{l=0}^{L-1} E(A_l)/T_l < 1$ and $T_l > 0, E(A_{l,i}) \geq 0$

However we have used Liestman-Campbell's algorithm in Phase 1 of our distributed scheduling algorithm.

3.2.2 Our Proposed Distributed Scheduler

3.2.2.1 Collection of Unscheduled Primaries

Our actual contribution starts with what we call Phase 2 of the distributed algorithm. This phase traverses through the fault-tolerant schedule obtained after Phase 1, and a linked list of primaries, which were not initially scheduled at the individual nodes, is formed.

The linked list was implemented with fields as shown in Fig. 3.2. Each node in the distributed system creates such a list, representing the *source, level,* and the *execution time* of the unscheduled primary. The fields *EST* (*Earliest Start Time*) and *LET* (*Latest End Time*), represent the time interval within which the primary algorithm has to be executed, while the field *server* represents the node that has accommodated this primary in its schedule.

Lemma:

Let i be a node, and L_i be the linked list formed during the Phase 2 of the

Source	Level	Ex - Time	EST	LET	Server

Figure 3.2: The fields of the linked list element

algorithm. The number of elements in the linked list is given as, $N(L_i) \leq T_{L-1} \sum_{l=0}^{L-1} 1/T_l$

The algorithm is given below :

Phase 2 Algorithm :

For every node i and every level l, $0 \leq l < L$ of node i do the following steps :

Step 1 : Traverse every time interval $[k * T_l, (k+1) * T_l]$, where $0 \leq k \leq T_{L-1}/T_l$. Check for the existence of primary $P_{l,i}$.

Step 2 : If primary $P_{l,i}$ is not found in the time interval, append the information regarding $P_{l,i}$ and the time interval to the list L_i.

3.2.2.2 Communication

At the end of the Phase 2 execution, the linked lists are ready for all the individual nodes containing the unschedulable primaries. The linked list is now communicated between the nodes of the distributed system using the Phase 3 algorithm. In a distributed system of N nodes, the linked list of a node is communicated through a cycle to the neighboring node, given

that the nodes of the distributed system are numbered in some order. The node receiving the list parses through the linked list to check whether it can schedule any of the unscheduled primaries. This is done by checking for large enough empty slots in the schedule of individual nodes, which can accommodate a primary and the corresponding communication delay. The knowledge of the maximum communication delay between the origin of the linked list and the present node is assumed to be known. If an empty slot is obtained, then the primary is scheduled and the field *server* in the linked list is changed to the present node number.

Theorem:

An unscheduled primary $P_{l,\phi}$, $0 \leq l < L$ of node ϕ can be scheduled in another node ψ if

$$T_l\{1 - \sum_{j=0}^{l} (E(A_{j,\psi}) + \sum_{k=0}^{T_l/T_j} \delta_{jk\psi} E(P_{j,\psi}))/T_j\} \geq E(P_{l,\phi}) + C_{\phi\psi}$$

where

$\delta_{jk\psi} = 0$ if $P_{j,\psi}$ is not scheduled in $[k * T_j, (k+1) * T_j]$ and

$\delta_{jk\psi} = 1$ if $P_{j,\psi}$ is scheduled in $[k * T_j, (k+1) * T_j]$

$C_{\phi\psi}$ = the round trip communication delay between ϕ and ψ

Proof :

Time slots occupied by alternates in node ψ up to level l is given as :

A = $T_l/T_0 * E(A_{0,\psi}) + ... + T_l/T_l * E(A_{l,\psi})$

Time slots occupied by primaries in node ψ upto level l is given as :

B = $(\sum_{r=0}^{T_l/T_0} \delta_{0r\psi}\, T_l/T_0) * E(P_{0,\psi}) + ... + (\sum \delta_{lr\psi}\, T_l/T_l) * E(P_{l,\psi})$

Empty slots available for schedule is $T_l - (A + B)$. For schedulability of $P_{l,\phi}$ we must have:

$T_l - (A + B) \geq E(P_{l,\phi}) + C_{\phi\psi}$

After scheduling the maximum possible primaries from the linked list,

the node sends the list to the next node. The communication is completed, once around the cycle, when each node receives back its own linked list. The linked list is traversed by the present node to determine various nodes that have scheduled the unschedulable primaries.

Therefore, while the unscheduled primaries of the individual nodes are communicated to the other nodes in the system at pre-runtime, the corresponding alternates are made sure to be executed in the individual nodes. If the result of a primary could not be communicated due to unexpected delay, an acceptable service can still be provided by executing the corresponding alternate algorithm in the individual nodes. Thus the distributed algorithm ensures better results without disturbing timing accuracy.

The new schedule contains more primaries, as compared to Phase 1 schedule, to be executed by various nodes. Since no existing primaries of the individual nodes are altered, the schedule remains Fault-Tolerant Optimal. But at the same time, the distributed algorithm uses the empty slots in the schedule of various nodes to schedule the unschedulable primaries of other nodes. Thus, at the least, when no empty slots are available, the distributed algorithm schedules the same number of primaries as the Liestman-Campbell's algorithm. But whenever large enough empty slots are found, the distributed algorithm schedules more primaries in different nodes.

The algorithm is given below :

Phase 3 Algorithm :

Step 1 : Each node i sends its linked list to the neighboring node j through the cycle.

Step 2 : Parse the schedule of node j for empty slot in the time interval

specified in every list element.

Step 3 : Calculate the communication delay between the origin and node j.

Step 4 : If (Execution time + round trip communication delay between the origin and node j) can be accommodated in the empty slot, then schedule one of the unscheduled primaries.

Step 5 : If scheduled then make (Empty time - (Execution time + Communication delay)) to be the new empty slot.

Step 6 : Change the *server* field of the linked list on the origin node to j to indicate that the job had been scheduled in node j and continue until, either all the list elements have been scheduled, or there is no empty slot for the jobs remaining in the list.

3.2.3 Dynamic Scheduling

3.2.3.1 Introduction

Thus far, a distributed fault-tolerant scheduler has been introduced for the nodes connected in a general network. The algorithm contains three phases, where the Phase 1 algorithm gives the initial schedule for individual nodes, the Phase 2 algorithm forms the linked list of all the unscheduled primaries in the individual nodes, and the Phase 3 algorithm communicates this list with other nodes and tries to schedule some of the unscheduled primaries in them.

The node from which the unscheduled primary originated is called the *Source* node and the alien node which schedules it is known as the *Server*

node. At the end of the Phase 3 algorithm, every node has received back its linked list, showing where some of its primaries have been scheduled.

During actual execution of the jobs, if a primary succeeds in its execution, i.e., if the primary meets the deadline, the execution of the corresponding alternate scheduled after the primary for the same level becomes redundant. We propose a dynamic fault-tolerant scheduler such that if a primary succeeds then the dynamic scheduler tries to schedule one of the unscheduled primaries in place of the redundant alternate.

3.2.3.2 Table Formation

Selection of a primary to be scheduled in the place of a redundant alternate could cause serious run-time overhead. The selection must be done in the shortest possible time. In order to minimize the overhead, every node should possess information regarding the unscheduled primaries from all other nodes of the system. A table containing such information is therefore created using the linked list received after Phase 3 algorithm. The table is filled with the execution times, (ETs), of the unscheduled primaries. This list of unscheduled primaries in the source is broadcast to all the nodes.

A global table containing information regarding all the unscheduled primaries of the system is created by every node using the linked list received from other nodes.

A shortest_job list, containing the unscheduled primaries using the global table is then created. This list is sorted in ascending order of (ET+CD), where CD is the communication delay. The lists formed for higher levels are appended to the lists formed for lower levels. A copy of shortest_job list is maintained so that the changes to the list are done not on the original list but on the working list.

Lemma:

The number of list elements in the shortest_job list, for any time interval of a node is bounded by $NL - \sum_{i=0}^{N-1} \sum_{l=0}^{L-1} \sum_{j=0}^{(T_{L-1}/T_l)-1} \delta_{lji}$

Lemma:

The number of unscheduled primaries in the system after Phase 3 is given by $N * T_{L-1} \sum_{l=0}^{L-1}(1/T_l) - \sum_{i=0}^{N-1} \sum_{l=0}^{L-1} \sum_{j=0}^{(T_{L-1}/T_l)-1} \delta_{lji}$

Proof :

Total number of primaries in a node is given by

$T_{L-1}/T_0 + T_{L-1}/T_1 + ... + T_{L-1}/T_{L-1}$

which is equal to $T_{L-1}(\sum_{l=0}^{L-1} 1/T_l)$

Therefore for *N-1* nodes we have

$(N\text{-}1 * T_{L-1})(\sum_{l=0}^{L-1} 1/T_l)$ number of primaries

Total number of primaries scheduled in n nodes in all levels is given by

$\sum\sum\sum(\delta_{lji})$

This difference gives the number of unscheduled primaries in the system.

3.2.3.3 A Dynamic Fault-Tolerant Schedule

The previous section illustrated how a shortest_job list is formed using the global table. The selection of the primary from shortest_job list to be scheduled in place of a redundant alternate is discussed in this section. A copy of the global table, namely the working table, is maintained and updated during runtime.

If a primary $P_{l,i}$, $0 \leq l < L$ of node i succeeds, then the execution of the corresponding alternate $A_{l,i}$ is redundant. In such a case, the first element of the working shortest_job list for the corresponding interval is checked for schedulability in the available time slot. The element is schedulable if the

$(ET + CD)$ of the primary in the working copy of shortest_job list is not greater than the (ET) of $A_{l,i}$.

In case the element in the working shortest_job list is schedulable, then the change of state is broadcast to other nodes in the system. Working tables and working lists are updated. Since the tasks are periodic, after execution of tasks in all levels of interval $[0..T_{L-1}]$, working lists and working tables are updated by the global shortest_job list and the global table, respectively.

By scheduling unscheduled primaries in the dynamically arising empty slots, the number of primaries scheduled are maximized. The purpose of creating a working shortest_job list was to minimize the overhead of selecting a primary at run-time when an alternate becomes redundant. In our algorithm, the scheduler dynamically picks the first element of the working list and schedules it if the timing constraints are satisfied.

The global_table and the global shortest_job list are stored for future references. The working table and working list are updated from the global_table and global_list after the time interval $[0..T_{L-1}]$. This is necessary because the system may not behave in the same manner during subsequent time intervals.

To avoid the overhead associated with the cost of code migration at run-time, codes for unscheduled primaries listed in the global data are broadcast to all nodes before run-time.

3.2.3.4 Algorithm

Make_Table_Column :
For every node i

Step 1 : Create the column elements such that they indicate the time interval $[j * T_l, (j+1) * T_l]$ where $0 \leq j \leq T_{L-1}/T_l$ for every $0 \leq l < L$

Step 2 : Traverse the schedule and for all the primaries scheduled, fill the respective column element with the present node number.

Step 3 : Traverse through the linked list received after the Phase 3 algorithm has scheduled all the primaries in other nodes, fill the respective column element with the node number where the primary has been scheduled.

Step 4 : Fill in the execution time of the unscheduled primary in the rest of the column elements.

Step 5 : Create a linked list of all the unscheduled primaries of step 4 with the details of source node, time interval, execution time and level.

Make_Global_Table :

Step 1 : Receive the linked list of unscheduled primaries from every node.

Step 2 : Create a global table with node numbers as the rows and the time intervals $[j * T_l, (j+1) * T_l]$ as the columns.

Step 3 : For every linked list received do :

Calculate the communication delay between the origin t of the linked list and present node i.

Fill in the row for node t with the sum of Execution time and Communication Delay on the respective columns containing the time intervals of levels.

Shortest job_list :

Step 1 : For every column $[j * T_l, (j+1) * T_l]$ create a list with fields containing source of primary, level and the sum of execution time and communication delay.

Step 2 : Append list $[j*T_l, (j+1)*T_l]$ with all the lists $[j*T_m, (j+1)*T_m]$ satisfying $(j+1)*T_m \leq (j+1)*T_l$ where $0 \leq j \leq T_{L-1}/T_l$, $0 \leq l < L$ and $0 \leq m \leq l$.

Step 3 : Sort by the sum of execution time and communication delay. The total number of lists are T_{L-1}/T_0

Broadcast codes of unscheduled primaries to all nodes
Run-Time Update of Table_Information :

Step 1 : If a message is received, then alter the corresponding table information using the fields source, and time-interval.

Step 2 : Delete the list element in the group of shortest_job list $[j*T_m, (j+1)*T_m]$ where $(j+1)*T_m \leq (j+1)*T_l$ for $0 \leq j \leq T_{L-1}/T_l$ and $0 \leq m \leq l$.

Chapter 4

System Configuration

We now discuss how our proposed algorithm can be applied to different system configurations such as the virtual ring and the binary n-cube interconnection networks.

4.1 Virtual Ring

4.1.1 Initial Schedule

We consider an exact simply periodic real-time system of N nodes connected in a virtual ring network. A fault-tolerant optimal schedule is found for each individual node using the Phase 1 algorithm described in the previous section.

In Figure 4.1, an example of the execution times for the primary and alternate algorithms for various levels in a distributed system of three nodes is shown. The arrival period of the periodic jobs to be serviced are 10, 20, and 40 with T_{l+1}/T_l being 2, $0 \leq l < L$ and the number of levels L being 3.

Figure 4.2 demonstrates the fault-tolerant optimal schedule obtained using the Phase 1 algorithm for the Deadline Mechanism. The arrow marks

	Level 0		Level 1		Level 2	
Node	P0	A0	P1	A1	P2	A2
0	9	9	1	1	2	2
1	5	2	8	5	1	1
2	5	4	8	6	9	8
Arrival Period	T0 = 10		T1 = 20		T2 = 40	

Figure 4.1: Execution time for an exact simply periodic system of 3 nodes

show pre-emption of the jobs.

4.1.2 Collection of Unscheduled Primaries

The fault-tolerant schedule obtained after Phase 1 for an individual node is represented as an array. Phase 2 of the distributed algorithm parses through this array and a linked list of primaries, which were not initially scheduled at the individual nodes of the virtual ring network is formed.

Figure 4.3 illustrates the linked list formed after the execution of the Phase 2 of the algorithm by the three nodes containing all their respective unschedulable primaries. Node 0 creates a list of 7 elements, with fields shown in Figure 3.2. Similarly, node 1 and node 2 create a list of 2 and 7 elements respectively. Note that the *server* field is initialized to -1 to indicate that the primary has not been scheduled as yet.

4.1.3 Communication in Virtual Ring

At the end of Phase 2 execution, the linked lists contain the unschedulable primaries for all the individual nodes. The linked list is now communicated between the nodes of the distributed system using the Phase 3 algorithm. In a distributed system of n nodes connected through a virtual ring network,

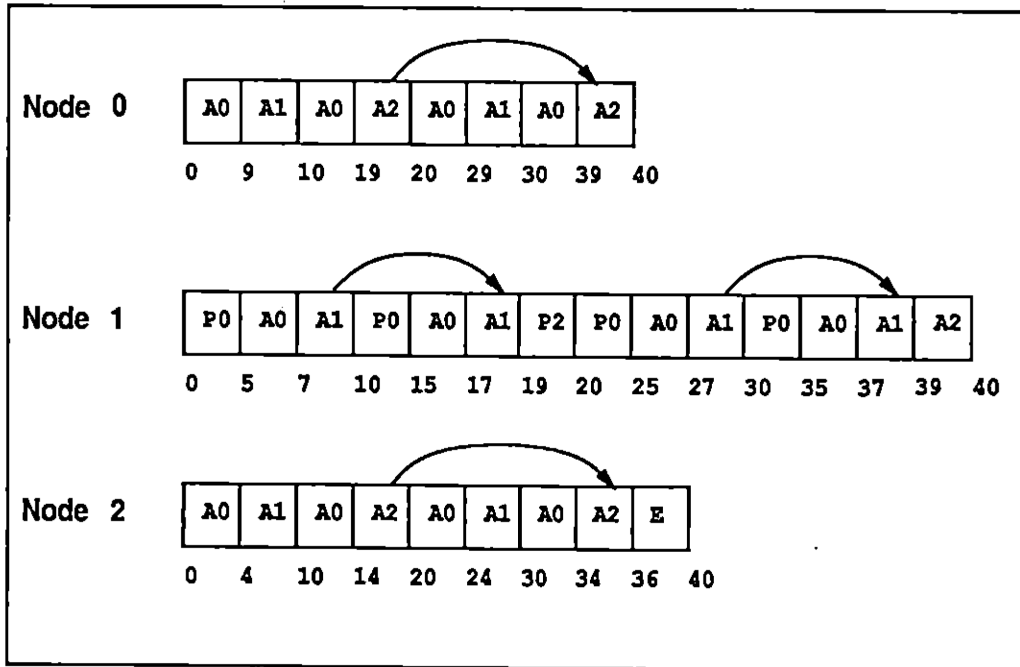


Figure 4.2: Schedule in individual nodes after Phase 1

the linked list of node i is communicated to the node $(i+1)\ mod\ N$, given that the nodes of the distributed system are numbered. The node receiving the list traverses through the linked list to check whether it can schedule the primaries. This is done by checking for empty slots in the schedule of individual nodes, which are large enough to accommodate the primary and the communication delay. The knowledge of the maximum communication delay between the origin of the linked list and the present node is assumed to be known. If an empty slot is obtained, then the primary is scheduled and the field *server* in the linked list is changed to the present node number.

After scheduling the maximum possible primaries from the linked list, the node sends the list to the next node. The communication is completed, once around the ring, when node i receives its own linked list. The linked list

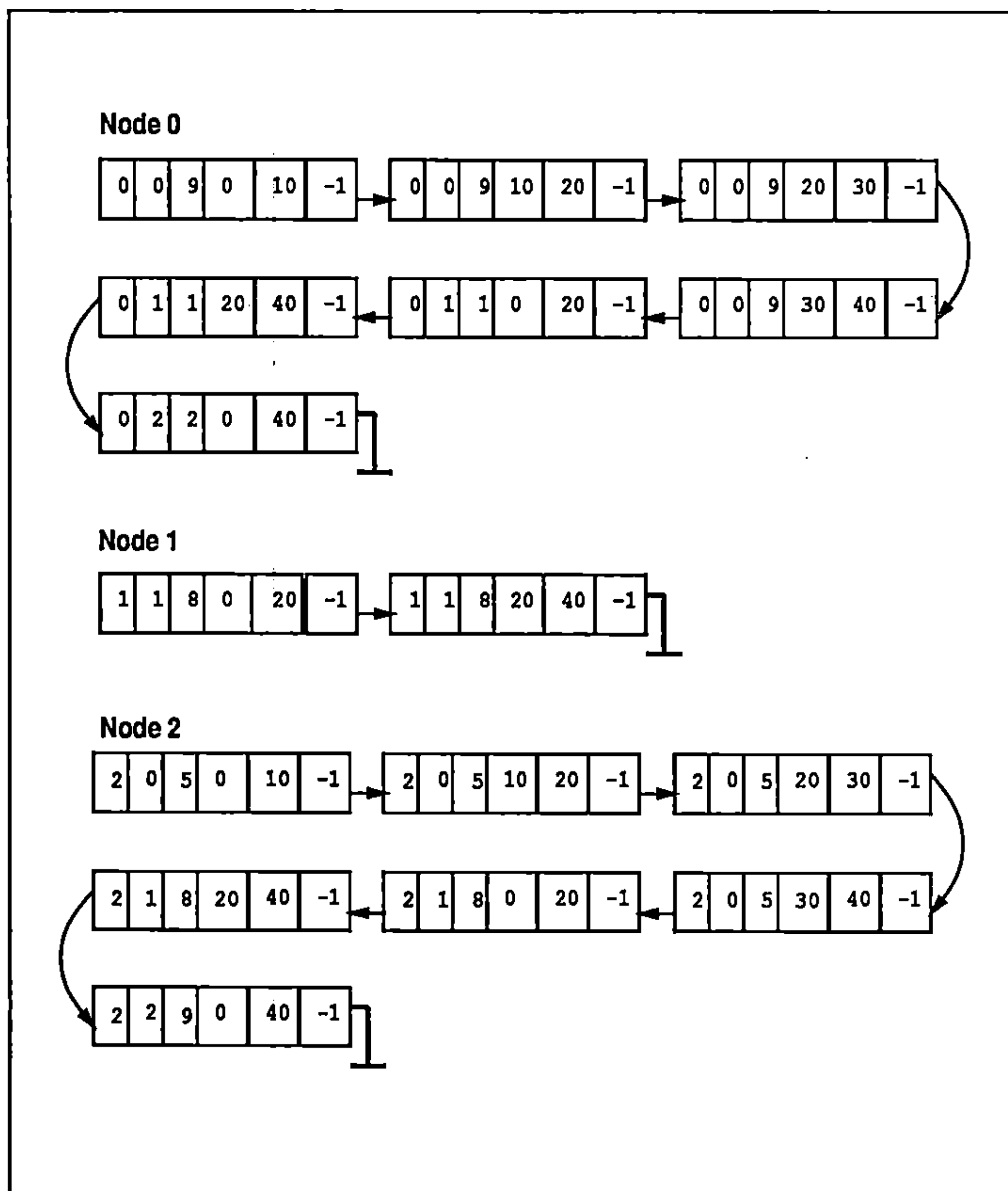


Figure 4.3: Linked list of unscheduled primaries

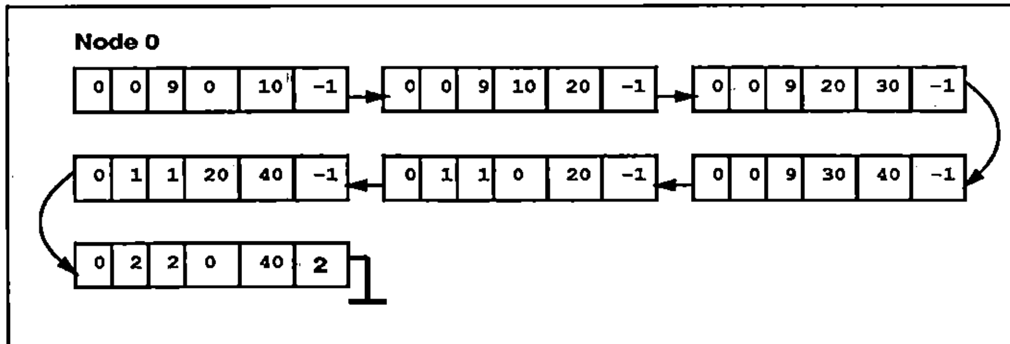


Figure 4.4: Linked list after communication with other nodes

is parsed by the present node to determine various server nodes that have scheduled the unschedulable primaries.

The linked list shown in Figure 4.4 is obtained after Phase 3 of the algorithm. Each node receives its own list back, with a modification on the *server* field of the list element. The figure shows a primary of node 0 scheduled at node 2. Thus node 0 waits for the result of this primary to arrive from node 2 during the time interval of execution.

Figure 4.5 demonstrates the new schedule for node 2, including the primaries of other nodes which have been scheduled in it. The schedules of node 0 and node 1 have not changed, but we note that the new schedule of node 2 includes an extra $P_{2,0}$ of node 0, being scheduled in the time interval 36-40.

4.1.4 Dynamic Scheduling in Virtual Ring

In this section, the dynamic scheduler introduced earlier is applied to the virtual ring network.

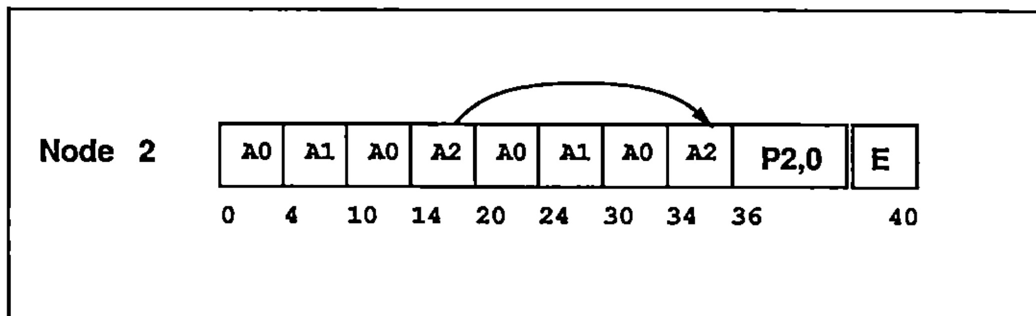


Figure 4.5: New schedule indicating some primaries of Node 0 scheduled at Node 2

4.1.4.1 Table Formation

Selection of a primary to be scheduled in place of a redundant alternate could cause serious run-time overhead. The selection must be done in an optimal time so that the node does not skip deadlines. In order to minimize the overhead, every node should possess information regarding the unscheduled primaries in every other node of the system. A table containing such information is created which holds the execution times (ETs) of the unscheduled primaries. Figure 4.6 represents the linked list of node 1 received back after traversing through the virtual ring using Phase 3 algorithm for the network. The list indicates that $P_{1,1}$ has not been scheduled. A linked list of the unscheduled primaries created using the table is illustrated in Figure 4.7. This list of unscheduled primaries in the source is broadcast to all the nodes.

A global table is created by every node using the linked list received from other nodes. Table entries correspond to node numbers and time intervals (TI) of various levels. Since $P_{0,1}$ is scheduled in node 1 for interval (0-10), the entry $(1, P_0(0-10))$ in Figure 4.8 shows an x mark. Otherwise, e.g., for node 2, the entry (2,P_0(10-20)) represents the sum of ET of $P_{0,2}$ and Communication Delay (CD) between node 2 and node 1. Thus a global

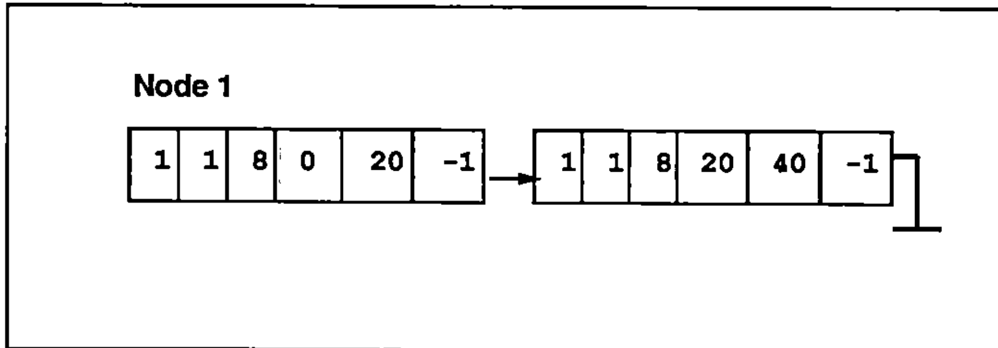


Figure 4.6: Linked list of node 1 after Phase 3 Algorithm

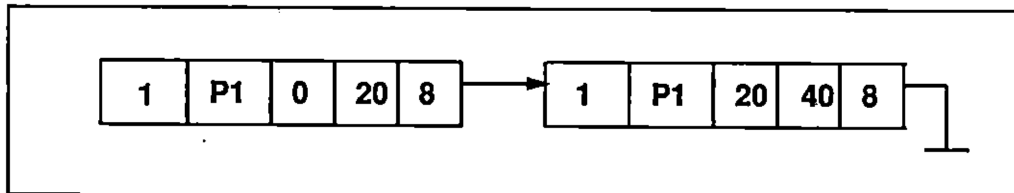


Figure 4.7: Linked list constructed using the initial table

table contains information regarding all the unscheduled primaries of the system.

Next, a sorted list of unscheduled primaries using the global table is created. Figure 4.9 illustrates the group of T_{L-1}/T_0 number of shortest_job lists for node 1. Each entry as illustrated indicates the node identifier, the primary of a level which is still unscheduled and the sum of ET and CD. A separate list is formed for each time interval, and sorted in ascending order of (ET + CD).

4.1.4.2 Run-Time Considerations

A copy of the global table, namely the working table, is maintained during execution to store changes node to the table during run-time. For example, as shown in figure 4.10, if the primary $P_{0,1}$ succeeds, then the alternate

N \ TI	P0 [0-10]	P0[10-20]	P0[20-30]	P0[30-40]	P1[0-20]	P1[20-40]	P2[0-40]
0	9+1	9+1	9+1	9+1	1+1	1+1	x
1	x	x	x	x	8	8	x
2	5+1	5+1	5+1	5+1	8+1	8+1	9+1

Figure 4.8: Global table constructed by node 1

$A_{0,1}$ becomes redundant. Consequently, the first element of the working shortest_job list for interval $(0-10)$ is checked for schedulability in the $(5-7)$ time slot. The element is schedulable if the $(ET+CD)$ of the first element $\leq (ET)$ of $A_{0,1}$. Since $(ET+CD)$ of primary $P_{1,0}$ of node $0 \leq (ET)$ of $A_{0,1}$, therefore $P_{1,0}$ of node 0 is scheduled in node 1.

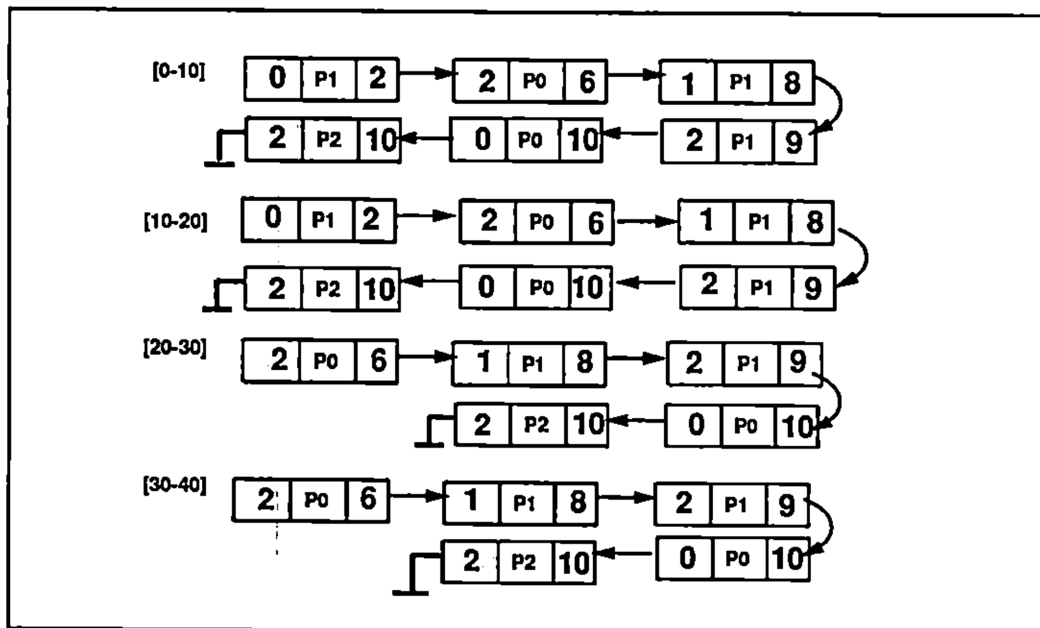


Figure 4.9: Group of sorted job lists for node 1

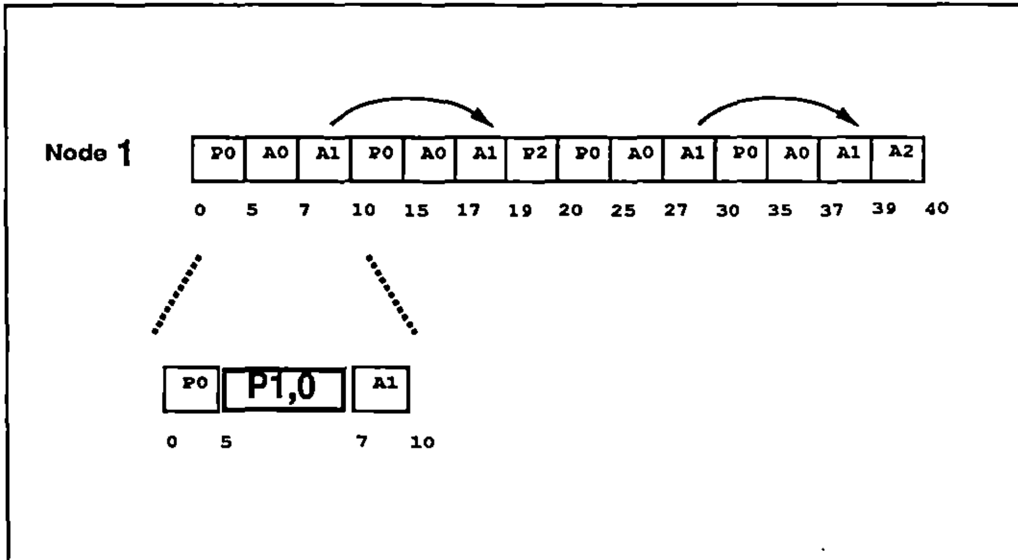


Figure 4.10: The updated schedule of node 1

The change of state is broadcast to other nodes, thus informing other nodes in the system that $P_{1,0}$ of node 0 has been scheduled in 1. Working tables are updated. Figure 4.11 illustrates the new working table for node 1. Since the tasks are periodic, after execution of tasks in all levels of interval $[0..T_{L-1}]$, the working list and the working table are updated by the global shortest_job list and the global_table respectively.

In applications where updating the table may compromise the timing constraints, the working table is not updated. In such cases, multiple results are sent to the source and the first (or any) one of the results is accepted.

N \ TI	P0 [0-10]	P0[10-20]	P0[20-30]	P0[30-40]	P1[0-20]	P1[20-40]	P2[0-40]
0	9+1	9+1	9+1	9+1	X	1+1	X
1	X	X	X	X	8	8	X
2	5+1	5+1	5+1	5+1	8+1	8+1	9+1

Figure 4.11: The updated working table for Node 1

4.2 Binary n-cube

4.2.1 Initial Schedule

Recent advances in VLSI have enabled the technical feasibility of large multicomputer networks such as n-cube architectures. We propose a fault-tolerant scheduler for nodes of a network connected as a binary n-cube. We consider an exact simply periodic real-time system of 2^n nodes connected in a binary n-cube interconnection network. An n-cube [73] is a generalization of a cube to n dimensions, where each of the 2^n nodes are connected to n nodes. Some of the commercially produced n-cube multicomputers are described in [74] [75]. The degree of a node and the diameter of a cube in this type of structure are equal to n.

Definition : An n-dimensional binary cube [73] is obtained by connecting each processor to n neighbors. Each node is labeled by an n-tuple $(b_0, b_1, ..., b_{n-1})$, where b_i is either 0 or 1. Two nodes are neighbors if they are joined by an edge, i.e., their co-ordinates agree in all but one of the n places.

The individual nodes of the n-cube use the Phase 1 algorithm for scheduling the jobs. A list containing the unschedulable primaries is formed by the individual nodes, to which the proposed algorithm is applied to sched-

Binary Equivalent	Node	Execution time, level 0,1,2 P0	A0	P1	A1	P2	A2
000	0	9	9	1	1	2	2
001	1	5	2	8	5	1	1
010	2	5	4	8	6	9	8
011	3	1	1	1	1	3	2
100	4	6	4	3	1	3	3
101	5	7	7	1	1	2	1
110	6	7	6	7	7	2	1
111	7	3	3	5	3	7	5

Figure 4.12: Execution time for various primary and alternate algorithms for nodes in 3-cube.

ule maximum primaries. If consider the communication delay between the neighboring nodes to be one unit, then any two nodes can communicate in maximum n units of time.

Figure 4.12 gives an example of the primary and alternate algorithm execution times chosen randomly for binary 3-cube.

Figure 4.13 illustrates the initial schedule obtained using Phase 1 algorithm for Binary 3-cube.

4.2.2 Collection of Unscheduled Primaries

The fault-tolerant schedule obtained after Phase 1 for an individual node is represented as an array. Phase 2 of the distributed algorithm traverses through this array and a linked list of primaries, which were not initially scheduled when the individual nodes were formed. Figure 4.14 illustrates the linked list formed by all the nodes of binary 3-cube. Node 011 forms

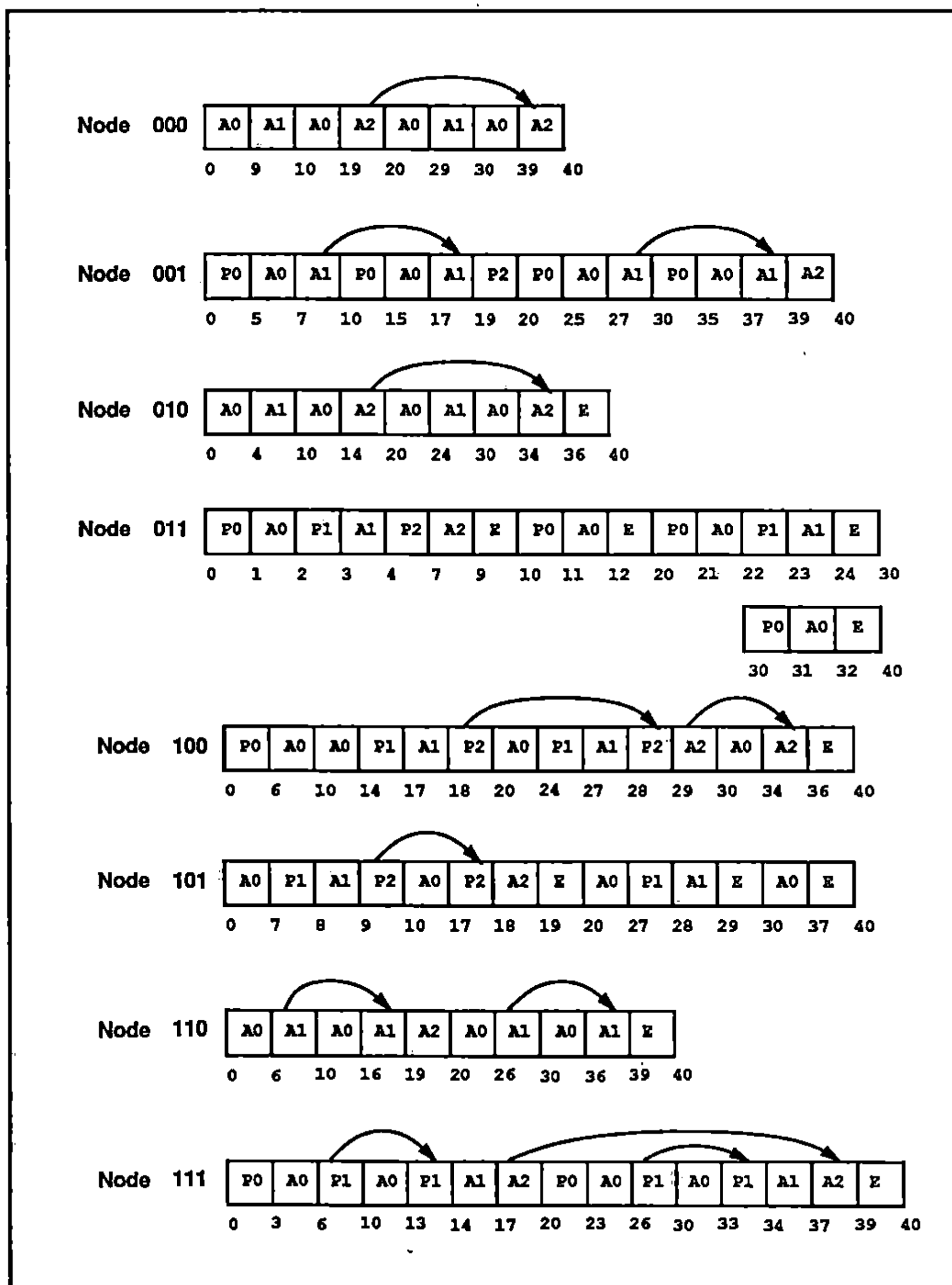


Figure 4.13: Scheduling individual nodes using Phase 1 .

a dummy list since all its primaries have been accommodated in its own schedule.

After the Phase 2 algorithm, the individual nodes of the binary n-cube are left with the linked list of unschedulable primaries. The Phase 3 algorithm is used for communicating the linked list of the nodes to the other nodes in order to schedule the unscheduled primaries if an empty slot exists.

4.2.3 Communication in Binary n-Cube

It is important to consider the communication delay between a *source* node and a *server* node in order to schedule the primary of the *source* in the *server* node. The *server* must find an empty slot equal to the sum of execution time of the primary and the communication delay between the *source* and the *server*. It is well known that if $\phi, \psi \in Q_n$, ϕ can communicate with ψ in n steps at the maximum.

Lemma:

Let $\phi(x_1, x_2, ..., x_{N-1})$ and $\psi(y_1, y_2, ..., y_{N-1})$ be any two processors in an n-cube. Then the communication delay between ϕ and ψ is given by

$C_{\phi\psi} = \sum_{i=1}^{n} \chi_i$

where

$\chi_i = 0$ if $x_i = y_i$

$\chi_i = 1$ if $x_i \neq y_i$

A 1 unit communication delay is assumed between neighboring nodes. The communication between the nodes has to be done in a way where the linked list of a source is sent to all the nodes in the n-cube to find a server which can execute one or more primaries from the linked list. One way of communicating the linked list to other nodes and not ending with duplicate servers is to find a *Hamiltonian cycle* in the n-cube. It can easily be shown

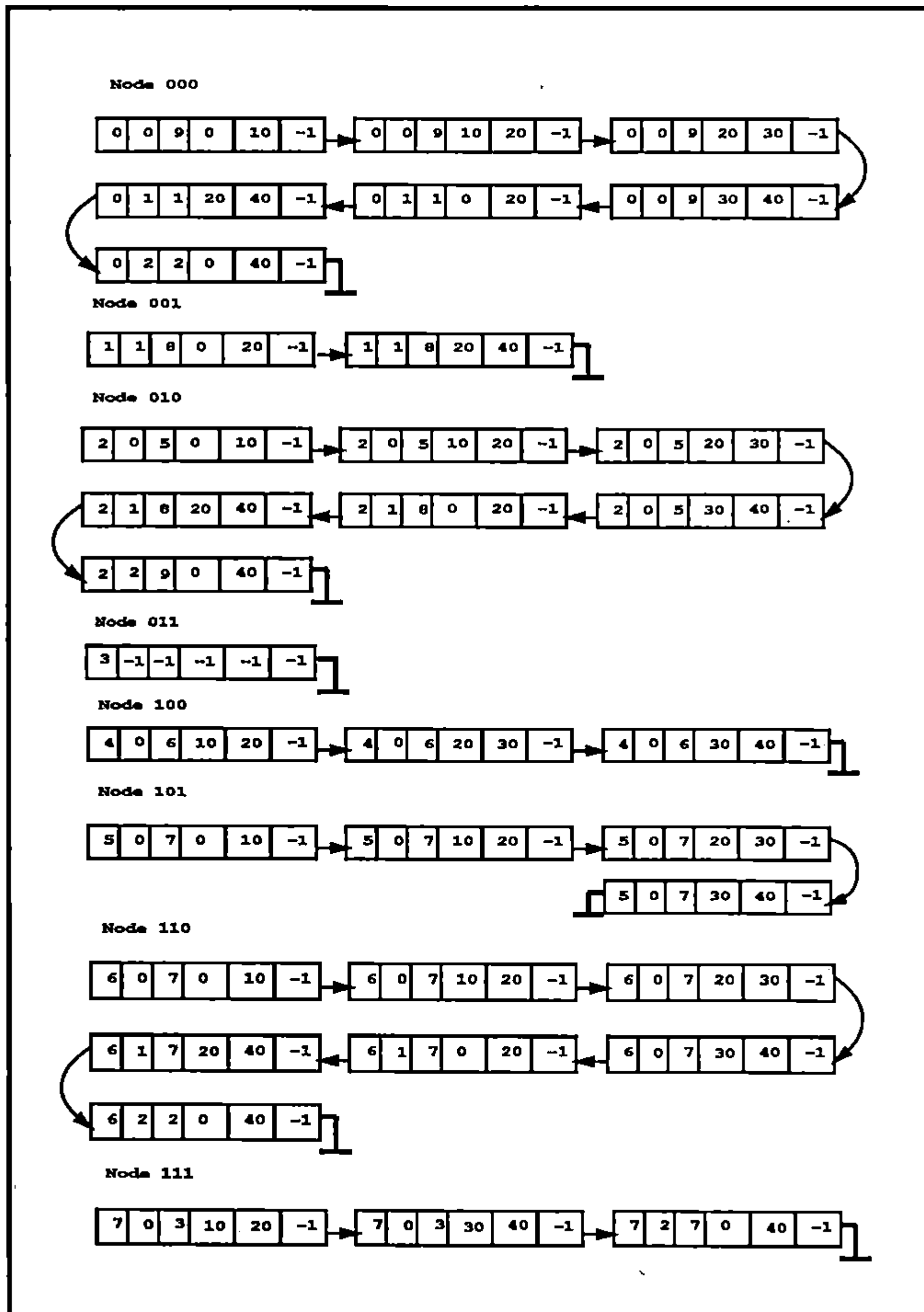


Figure 4.14: Linked list formed after Phase 2

that every hypercube Q_n, has a Hamiltonian cycle.

The list is communicated through this Hamiltonian cycle. Each node in the n-cube sends its list to its neighbor according to the sequence of nodes in the cycle. The neighboring node checks whether it can schedule any of the primaries in the linked list. This is done by checking for empty slots large enough for scheduling any of the primaries in the linked list along with communication delay.

A Gray Code technique [76] [77] [78] [79]is used to communicate the list to other nodes of the system in a Hamiltonian cycle so that all the nodes can parse through the linked list belonging to every other node. Normally, the starting code word is taken to be (00...0) although since the code is cyclic it does not matter. Every n-bit Gray code set gives a Hamiltonian cycle for an n-cube.

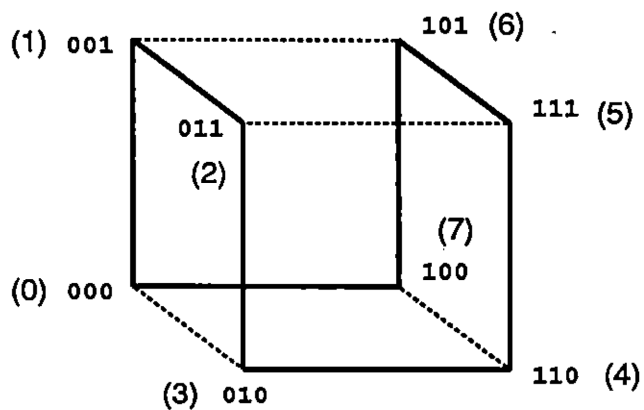
Lemma : In an ordered sequence of code words, if i = $b_n b_{n-1} ... b_0$ then codeword(i) = XOR(0,b_n) XOR(b_n,b_{n-1}) ... XOR(b_1,b_0).

Thus if all the 2^n nodes of the n-cube are ordered in some unique way, any individual node i with codeword(i), communicates with its neighbor j with codeword(i+1). Thus the communication is done in a Hamiltonian cycle. Figure 4.15 shows how an n-cube can be ordered, and a Hamiltonian cycle could be found using a Gray code. A node can obtain the code word of its neighbor using the node number that has been assigned to it. The numbers given within brackets at every node in the Figure 4.15 represents the node number of that node.

The communication is complete for a node i when it receives its own linked list back, after traversing once across all the nodes of n-cube. The linked list is parsed to determine various nodes that have scheduled the initially

Node (i)	Codeword (i)	Codeword (i+1)
000	000	001
001	001	011
010	011	010
011	010	110
100	110	111
101	111	101
110	101	100
111	100	000

Table giving the Hamiltonian cycle for 3-cube



The dark lines indicate the Graycode Hamiltonian cycle

Figure 4.15: Hamiltonian path in 3-cube and corresponding Gray code.

Node	Empty Slots before Phase 3 algorithm	Empty Slots after Phase 3 algorithm
000	None	None
001	None	None
010	36 - 40	36 - 40 P2,000 scheduled
011	09 - 10	09 - 10 Empty
	12 - 20	12 - 20 P0,100 scheduled
	24 - 30	24 - 30 P1,000 scheduled
	32 - 40	32 - 40 P0,100 scheduled
100	36 - 40	36 - 40 P0,111 scheduled
101	19 - 20	19 - 20 Empty
	29 - 30	29 - 30 Empty
	37 - 40	37 - 40 P2,110 scheduled
110	39 - 40	39 - 40 Empty
111	39 - 40	39 - 40 Empty

Figure 4.16: Empty slots before and after Phase 3 algorithm.

unschedulable primaries of the present node. Figure 4.16 illustrates the empty slots in the schedule before and after Phase 3 Algorithm in the 3-cube for the random data given in Figure 4.12. Figure 4.17 illustrates the new schedule for the random data. It can be clearly seen that six extra primaries have been scheduled.

The Phase 3 Communication algorithm is given below :

Phase 3 Algorithm :

Step 1 : Each node with node number i and code word codeword(i) sends

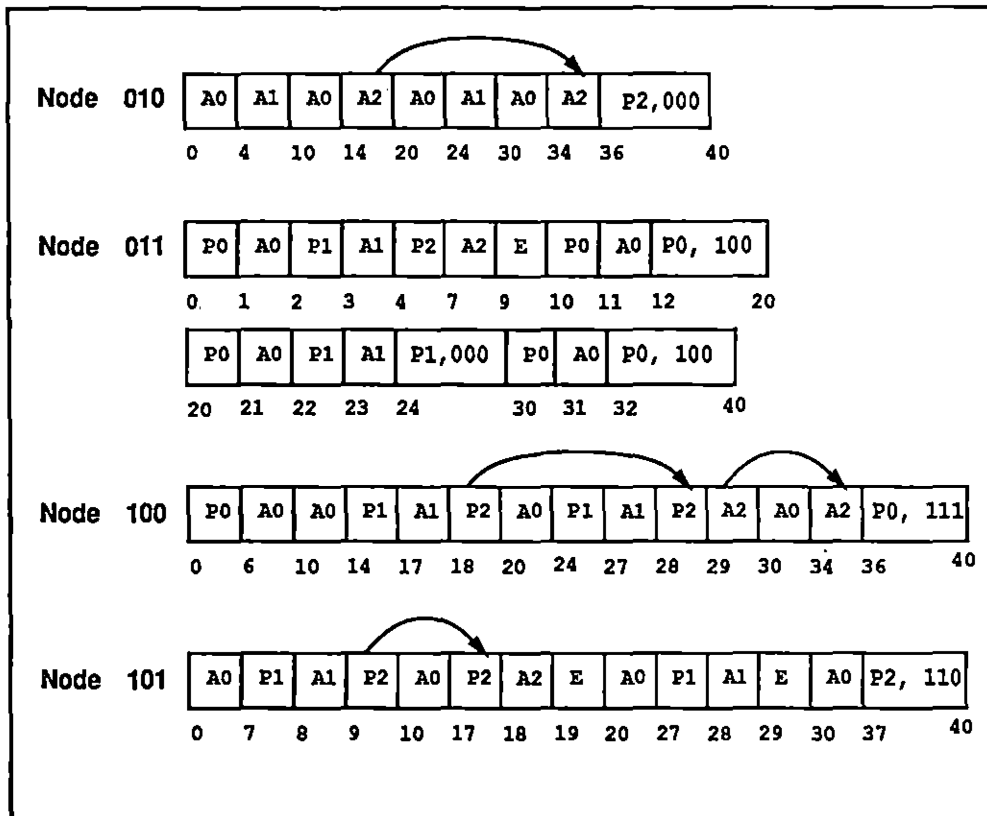


Figure 4.17: New schedule after Phase 3 algorithm.

its linked list to the neighboring node with node number j and code word codeword(i+1).

Step 2 : Parse the schedule of node j for empty slot in the time interval specified in every list element.

Step 3 : Calculate the communication delay between the origin and node j by finding the number of places the coordinates of binary representation of the origin and node j differ.

Step 4 : If (Execution time + Communication delay between the origin

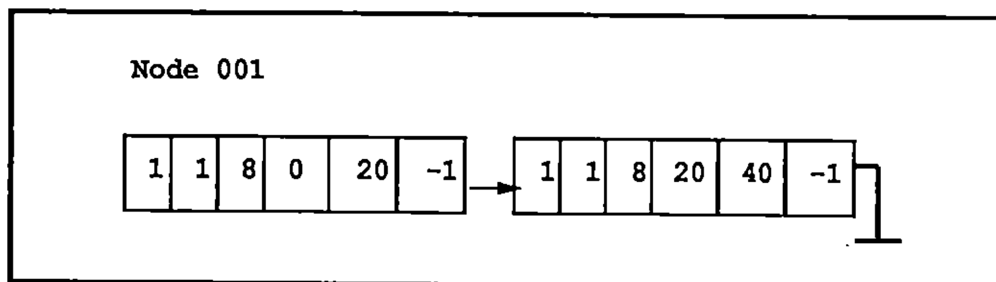


Figure 4.18: Linked List of node 001 after Phase 3 Algorithm

of an unscheduled primary and node j) can be accommodated in the empty slot then schedule the primary.

Step 5 : If scheduled then make (Empty time - (Execution time + Communication delay)) be the new empty slot.

Step 6 : Change the list entry to indicate that the job had been scheduled in node j and continue until either all the list elements have been scheduled, or there is no empty slot for the jobs remaining in the list.

4.2.4 Dynamic Scheduling in Binary n-Cube

4.2.4.1 Table Formation

We now consider the table formation needed for our 3-cube example. Figure 4.18 represents the linked list of unscheduled primaries of node 001 received back after traversing through *Hamiltonian* cycle using the Phase 3 algorithm. Figure 4.19 shows list of unscheduled primaries in the source node 001 which is broadcast to all the nodes.

Different methods for broadcasting in n-cube exist [80]. Broadcast in a binary n-cube could be done in n steps such that each node either receives multiple copies of the list or a single copy.

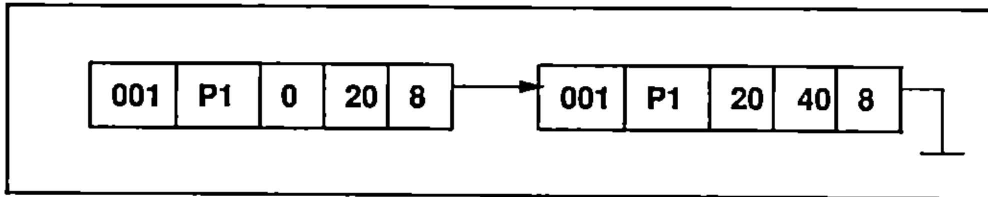


Figure 4.19: Linked list constructed using the initial table

N \ TI	P0 [0-10]	P0[10-20]	P0[20-30]	P0[30-40]	P1[0-20]	P1[20-40]	P2[0-40]
001	x	x	x	x	8	8	x
010	5+2	5+2	5+2	5+2	8+2	8+2	9+2
011	x	x	x	x	x	x	x
100	x	x	6+2	x	x	x	x
101	7+1	7+1	7+1	7+1	x	x	x
110	7+3	7+3	7+3	7+3	7+3	7+3	x
111	x	3+2	x	x	x	x	7+2
000	9+1	9+1	9+1	9+1	1+1	x	x

Figure 4.20: Global table constructed by node 001

A global table shown in figure 4.20 is created by every node using the linked list received from other nodes. Figure 4.21 illustrates the group of T_{L-1}/T_0 number of shortest_job lists for node 001 which was created using the global table.

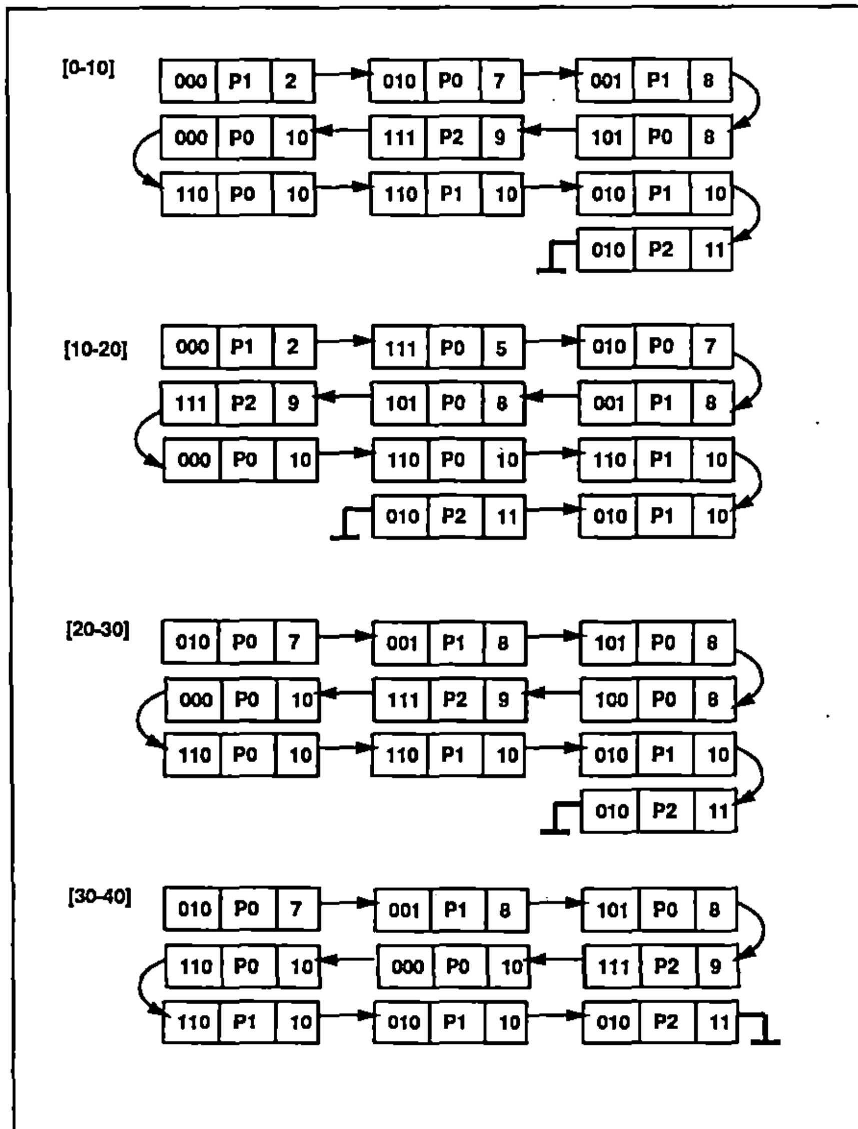


Figure 4.21: Group of sorted job lists for node 001

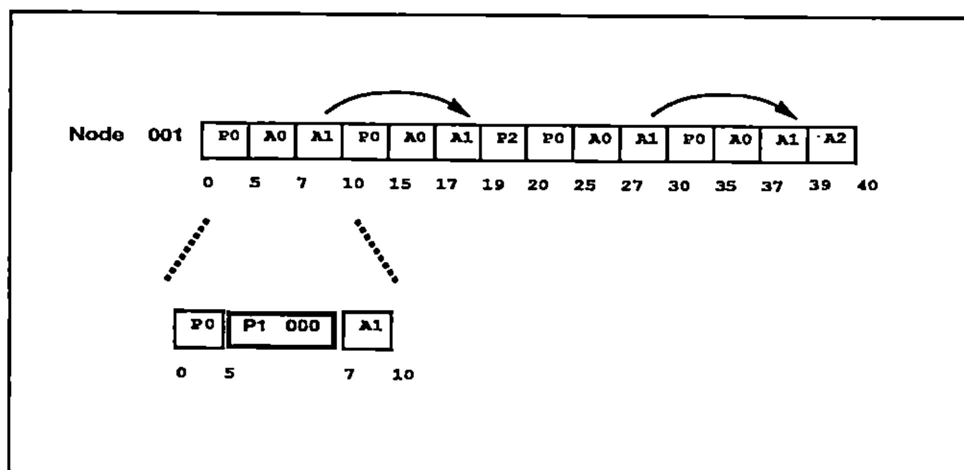


Figure 4.22: The updated schedule of node 001

4.2.4.2 Run-time Scheduling

As stated previously, if a primary $P_{l,i}$, $0 \leq l < L$ of node i succeeds, then the execution of the corresponding alternate $A_{l,i}$ is redundant. Figure 4.22 illustrates this for node 001. If primary $P_{0,001}$ succeeds in the interval $[0-10]$, the corresponding alternate $A_{0,001}$ scheduled in $[5-7]$ becomes redundant. The first element of the working shortest_job list for interval $[0-10]$ is checked for schedulability in the $[5-7]$ time slot. The element is schedulable if the $(ET + CD)$ of the first element in working shortest_job list is not greater than (ET) of $A_{0,001}$. Figure 4.23 illustrates that primary $P_{1,000}$ of node 000 has an (ET+CD) not greater than (ET) of $A_{0,001}$. Thus $P_{1,000}$ of node 000 is scheduled in 001.

The change of state is broadcast to other nodes and working table and shortest_job list are updated.

TI / Node	P0 [0-10]	P0[10-20]	P0[20-30]	P0[30-40]	P1[0-20]	P1[20-40]	P2[0-40]
001	x	x	x	x	8	8	x
010	5+2	5+2	5+2	5+2	8+2	8+2	9+2
011	x	x	x	x	x	x	x
100	x	x	6+2	x	x	x	x
101	7+1	7+1	7+1	7+1	x	x	x
110	7+3	7+3	7+3	7+3	7+3	7+3	x
111	x	3+2	x	x	x	x	7+2
000	9+1	9+1	9+1	9+1	X	x	x

Figure 4.23: The updated working table of Node 001

Chapter 5

Performance Evaluation

5.1 Introduction

In this chapter we present the simulation results for our proposed distributed fault-tolerant scheduler. The performance of the static scheduler is evaluated when applied to virtual-ring and binary n-cube networks.

5.2 Generation of Test Cases

The input to the scheduler is obtained from a random number generator which generates the execution times of the primary and alternate algorithms. The random numbers for the performance evaluation of the scheduler are generated from the *rand()* function of UNIX 4.1c OS [1]. The generator can be tested empirically by applying various statistical tests. A known distribution, namely, chi-square is used for comparison with results obtained through the performance of various statistical tests on the random number generator [81] [82]. Exhaustive tests for the distribution of the random numbers generated using UNIX 4.1c OS has been conducted by Sivakumar [83]. From the obser-

[1] $UNIX^{TM}$ Trademark of Bell Laboratories

vations it has been concluded that the distribution tends to be exponential. The exponential distribution is well suited for the simulation of the execution time for jobs [81] and thus, was used for simulating the proposed scheduler.

5.3 Results of Simulation for Virtual Ring

The scheduler for the virtual ring network was simulated for varied number of jobs. The execution times of the primary and alternate algorithms for various levels were generated. The arrival period of the lowest level, T_0 was taken to be 10 time units. The ratio between the arrival periods for jobs of various levels, T_{l+1}/T_l, was taken to be 2. Thus the arrival periods of jobs of various levels follow the pattern of 10, 20, 40, 80, ... The performance of the scheduler for various nodes and for various levels have been discussed in the following sections.

5.3.1 Performance for Two Levels in Virtual Ring

Every node in the virtual ring contains two levels. The jobs to be scheduled by the Phase 1 algorithm for an individual node i are $\{(P_{0,i}, A_{0,i}), (P_{1,i}, A_{1,i})\}$. Thus there are three primaries to be scheduled in each node, two $P_{0,i}$s and one $P_{1,i}$. Therefore, for a virtual ring of three nodes, there is a maximum of nine primaries to be scheduled. Figure 5.1 illustrates the performance of the scheduler after the Phase 1 algorithm and after the Phase 3 algorithm for a virtual ring of three nodes. The increase in the number of primaries scheduled by Phase 3 algorithm can be noticed. Similarly, figures 5.2, 5.3 and 5.4 illustrate the performance of the scheduler after the Phase 1 and after the Phase 3 algorithms for a virtual ring of four, five and six nodes respectively. It is clearly visible from the figures that, as the number of nodes

in the virtual ring increases, the number of additional primaries scheduled by Phase 3 increases. This is demonstrated by figure 5.5, where the number of extra primaries scheduled by the proposed distributed scheduler is illustrated for a virtual ring of nodes three, four, five and six.

5.3.2 Performance for Three Levels in Virtual Ring

The jobs to be scheduled in node i are $\{(P_{0,i}, A_{0,i}), (P_{1,i}, A_{1,i}), (P_{2,i}, A_{2,i})\}$. Thus there are seven primaries to be scheduled in each node, namely, four $P_{0,i}$, two $P_{1,i}$ and one $P_{2,i}$. Therefore, for virtual ring of three, four, five and six nodes, there are a maximum of 21, 28, 35 and 42 primaries respectively to be scheduled. Figure 5.6 illustrates the performance of the scheduler after the Phase 1 algorithm and after the Phase 3 algorithm for a virtual ring of three nodes. The increase in the number of primaries scheduled by the Phase 3 algorithm can be seen. Similarly, Figures 5.7, 5.8, and 5.9 illustrate the performance of the scheduler after Phase 1 and Phase 3 algorithms for virtual ring of four, five and six nodes respectively. It is clearly visible from the figures that, as the number of nodes in the virtual ring increase, the number of additional primaries scheduled by Phase 3 increase. This is ascertained by the Figure 5.10, where the number of extra primaries scheduled by the proposed distributed scheduler is illustrated for virtual ring of nodes three, four, five and six.

5.3.3 Performance for Higher Levels in Virtual Ring

Figures 5.11 and 5.12 represent the additional primaries scheduled on a virtual ring network for levels four and five respectively. The percentage of testcases where no additional primaries were scheduled for four levels, are

28%, 14%, 5% and 3% for virtual ring network of nodes three, four, five and six respectively. Our proposed distributed scheduler schedules the same number of primaries as Phase 1 algorithm in the worst case. Figure 5.13 illustrates the performance of the algorithm on average for various levels in a virtual ring network.

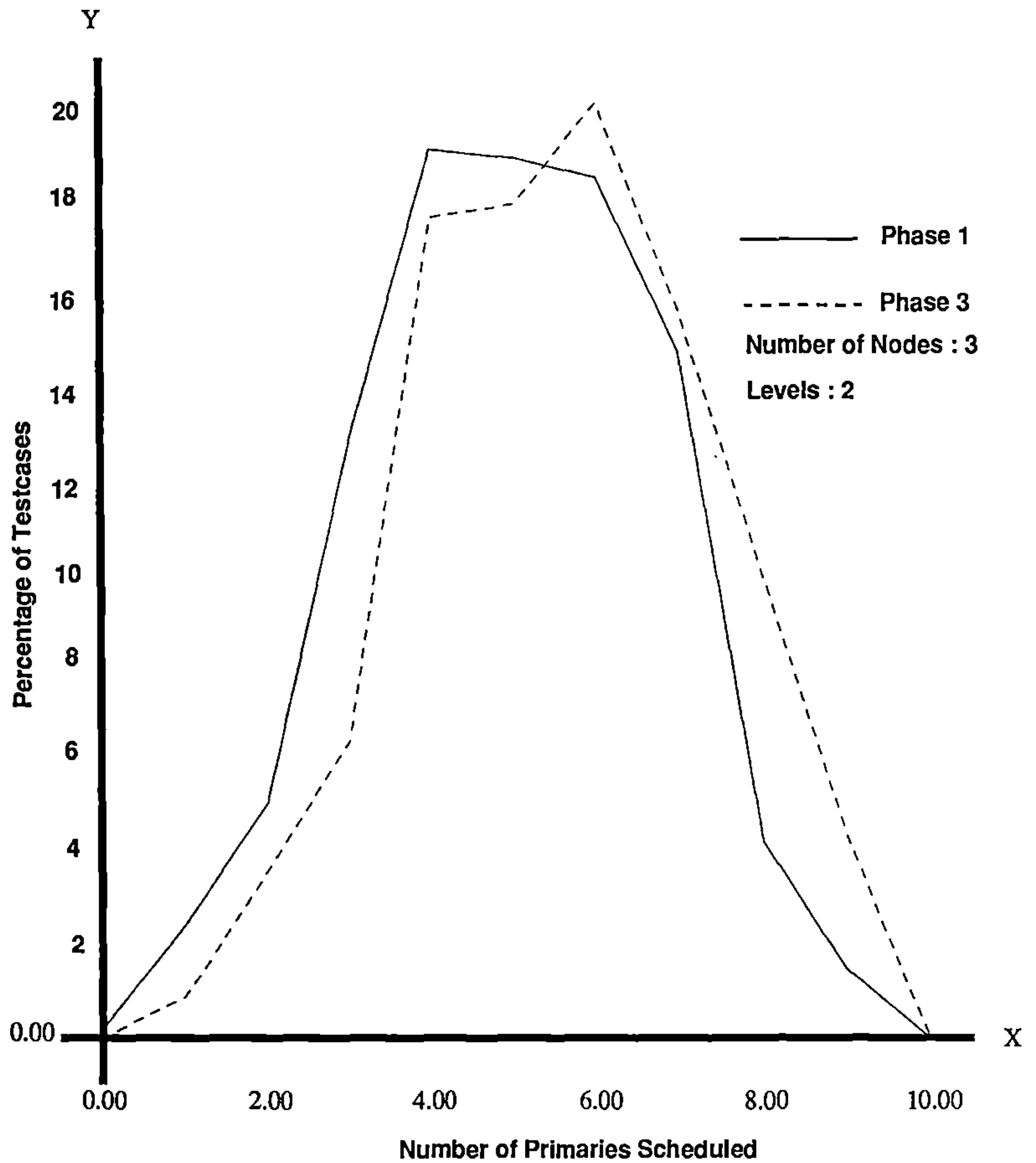


Figure 5.1: Performance of the scheduler for virtual ring of 3 nodes

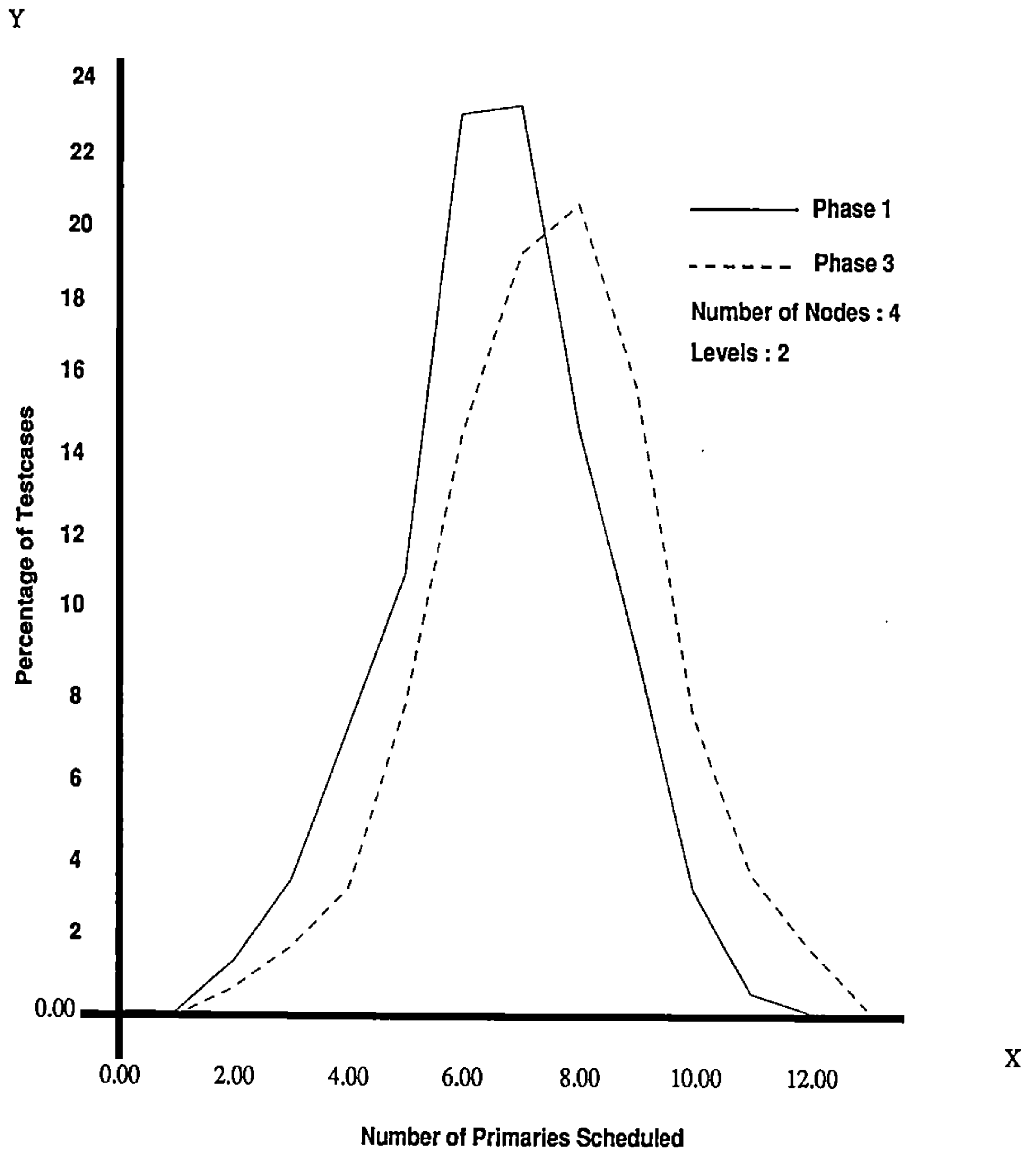


Figure 5.2: Performance of the scheduler for virtual ring of 4 nodes

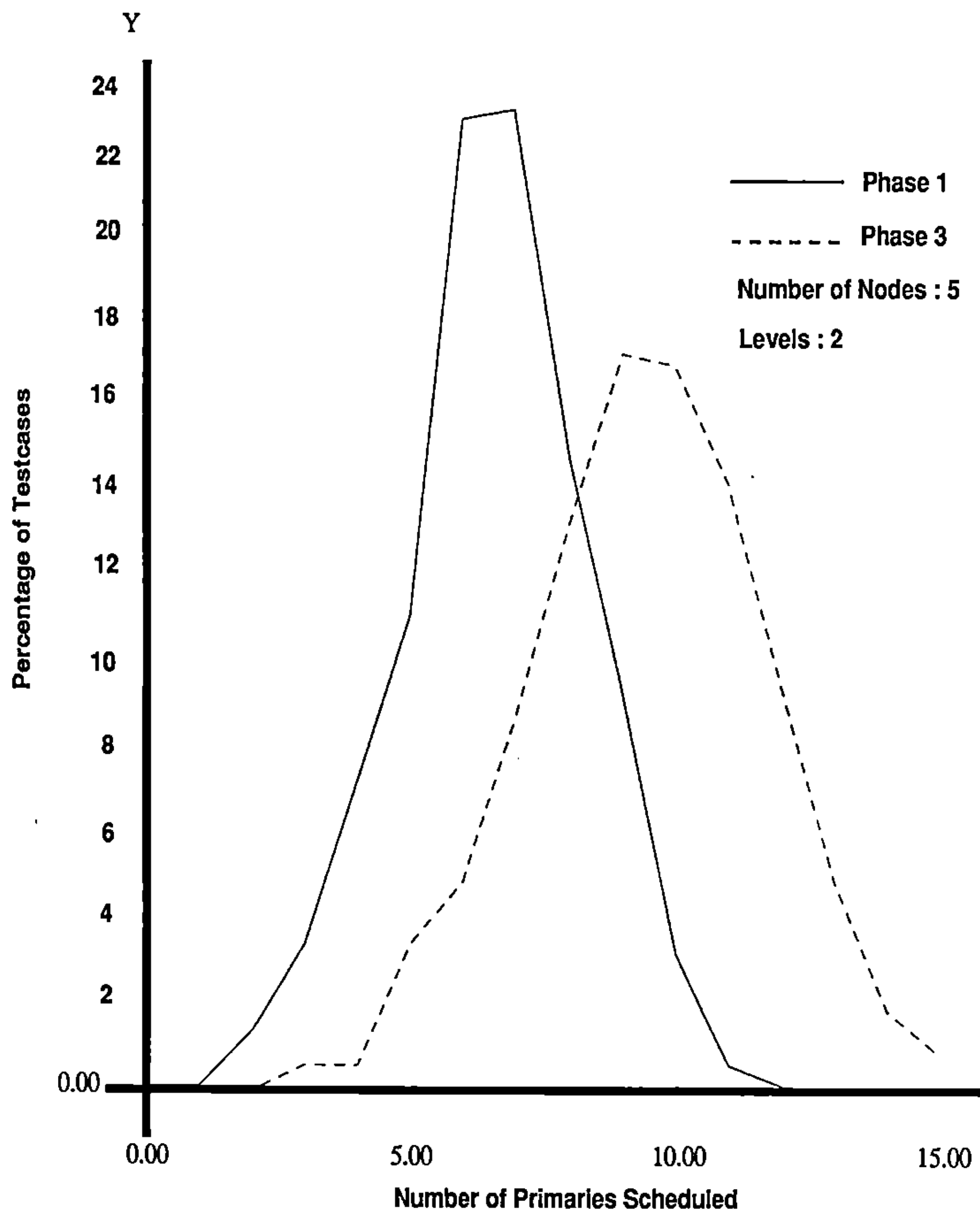


Figure 5.3: Performance of the scheduler for virtual ring of 5 nodes

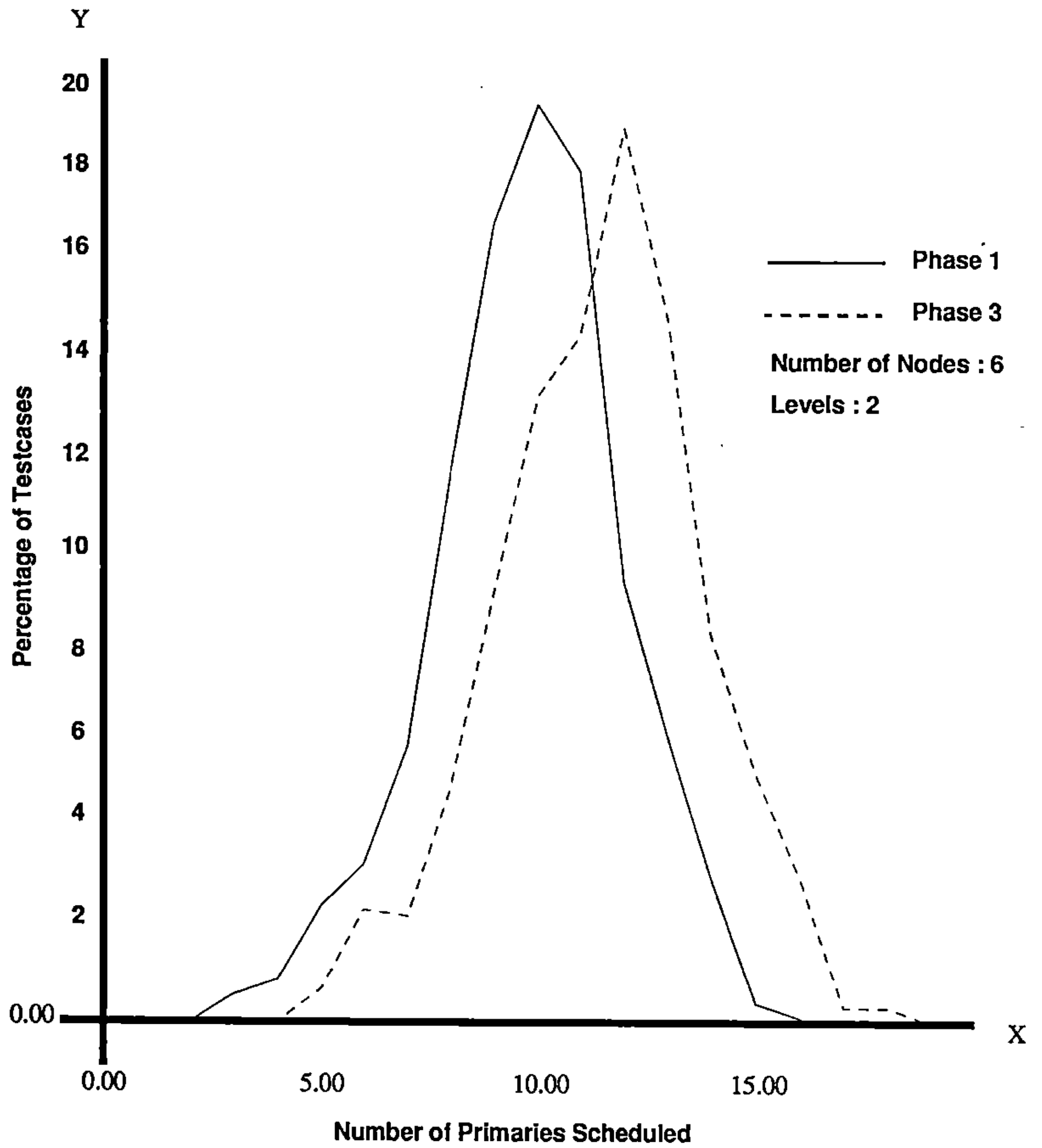


Figure 5.4: Performance of the scheduler for virtual ring of 6 nodes

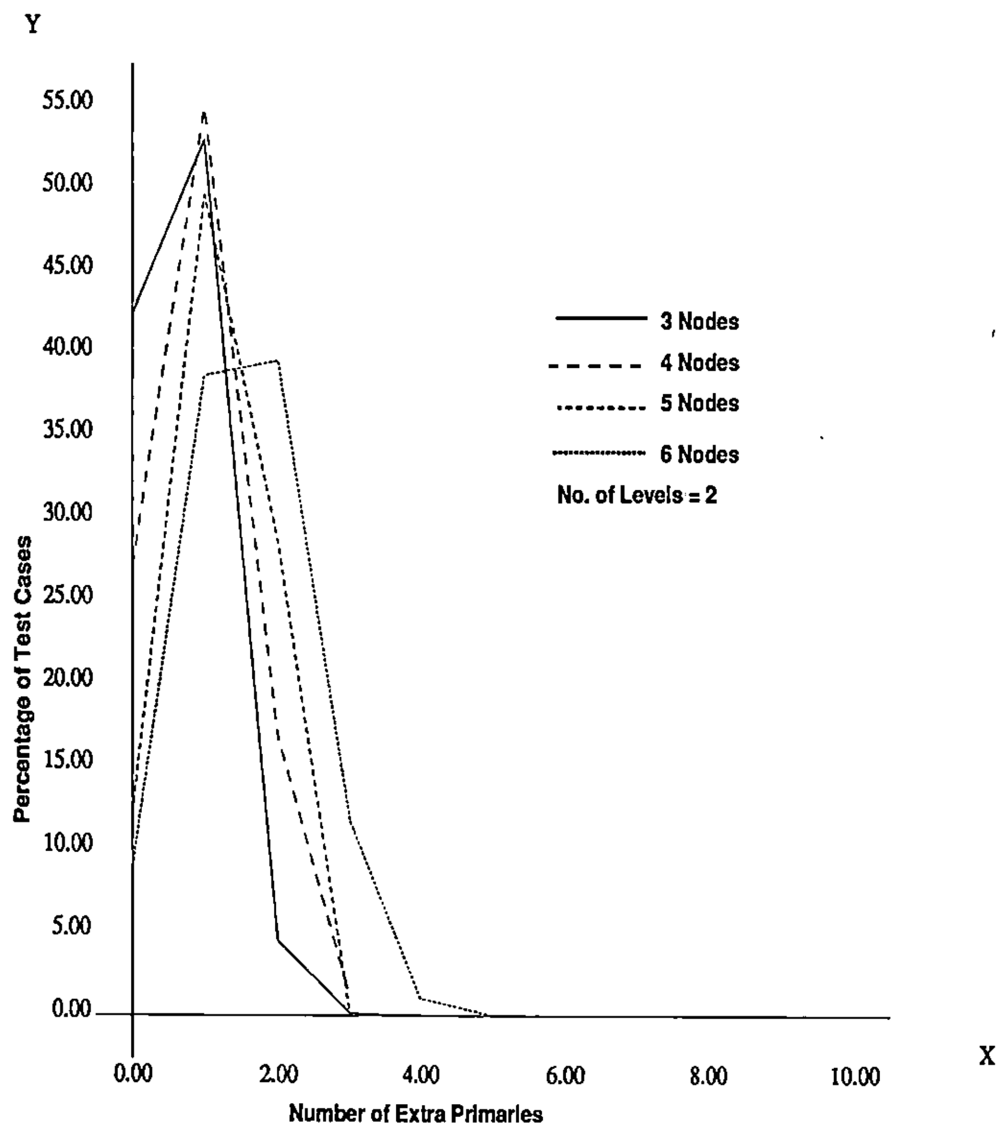


Figure 5.5: Percentage of test cases where extra primaries were scheduled for virtual ring network

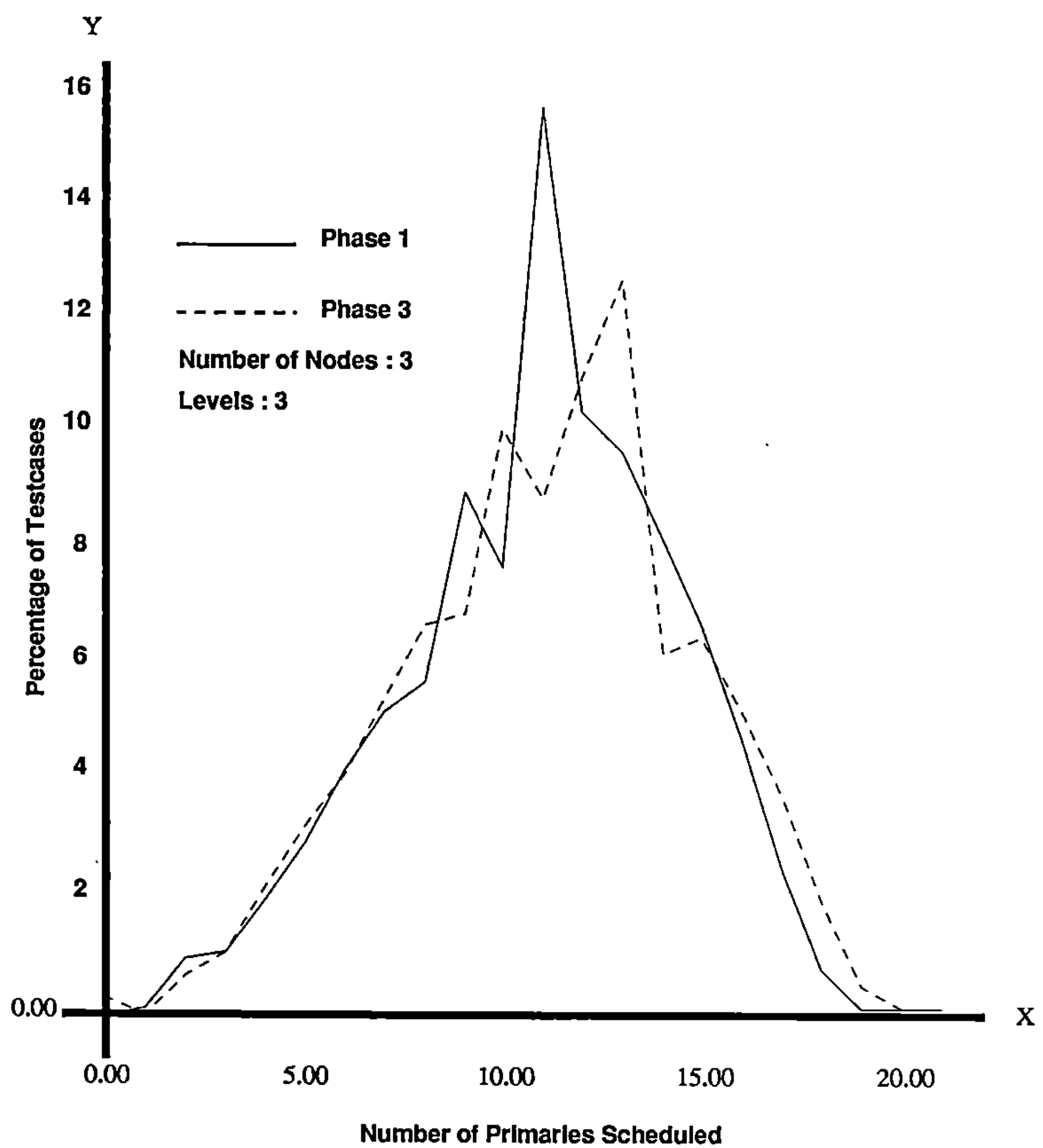


Figure 5.6: Performance of the scheduler for virtual ring of 3 nodes

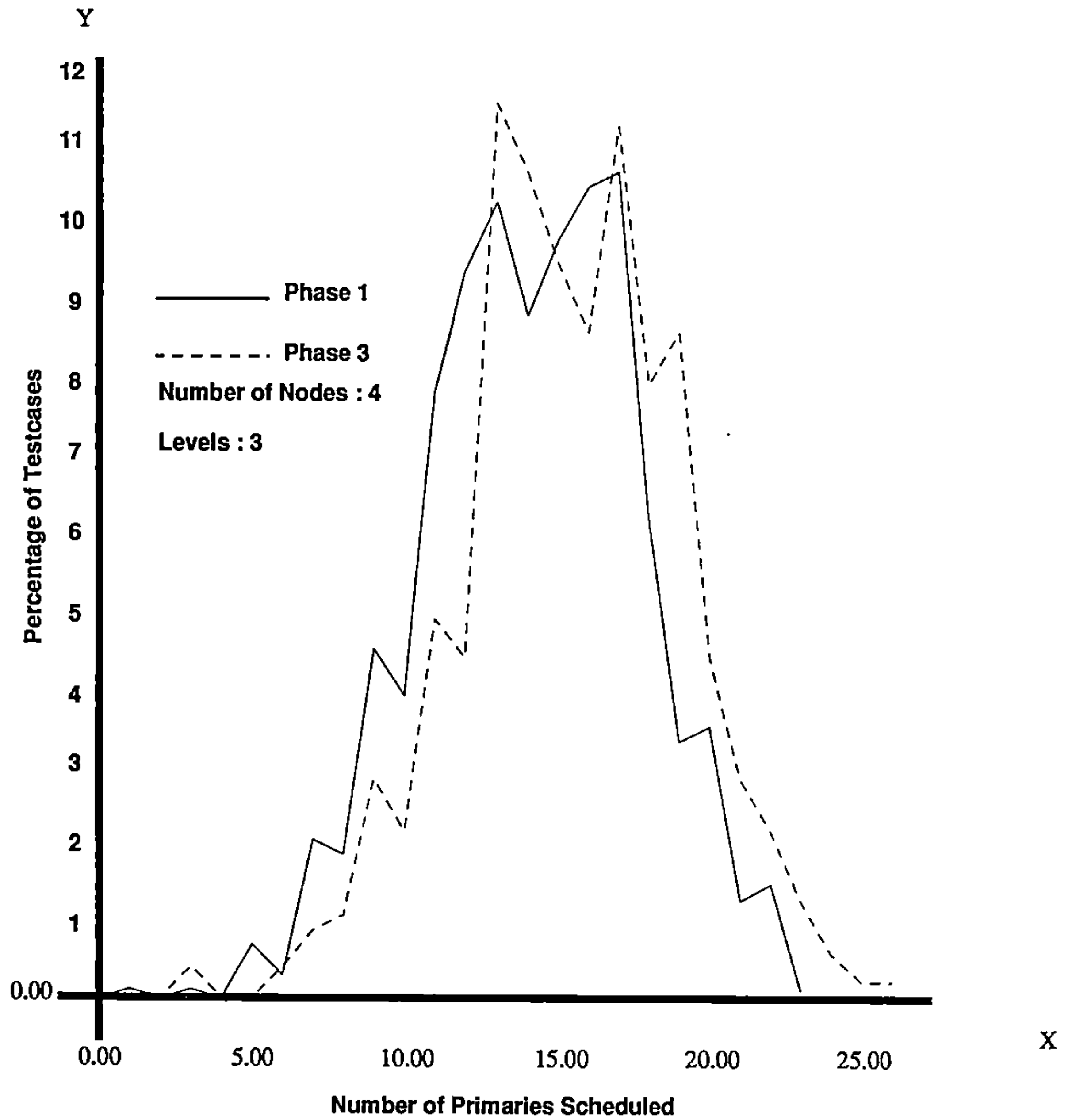


Figure 5.7: Performance of the scheduler for virtual ring of 4 nodes

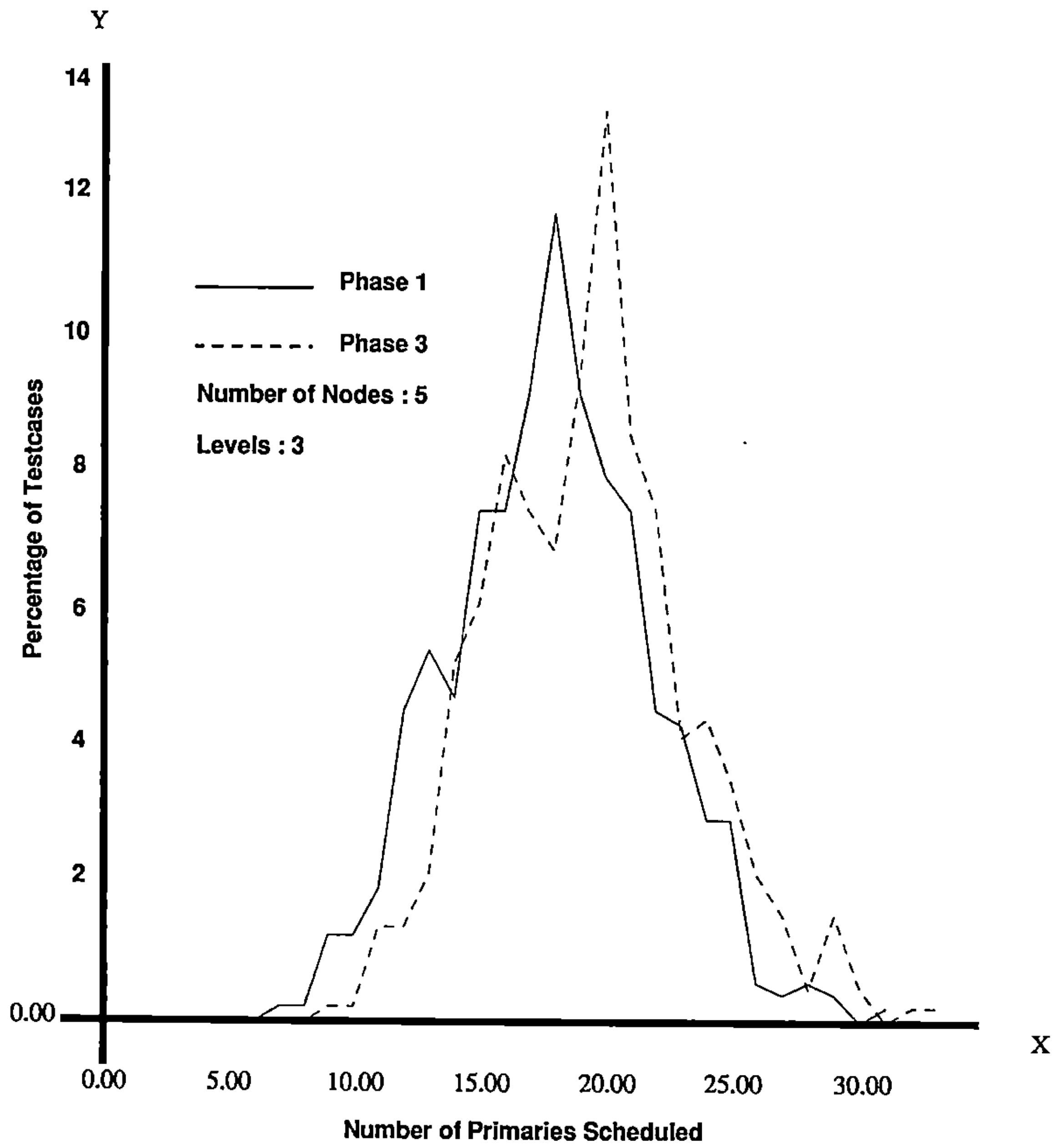


Figure 5.8: Performance of the scheduler for virtual ring of 5 nodes

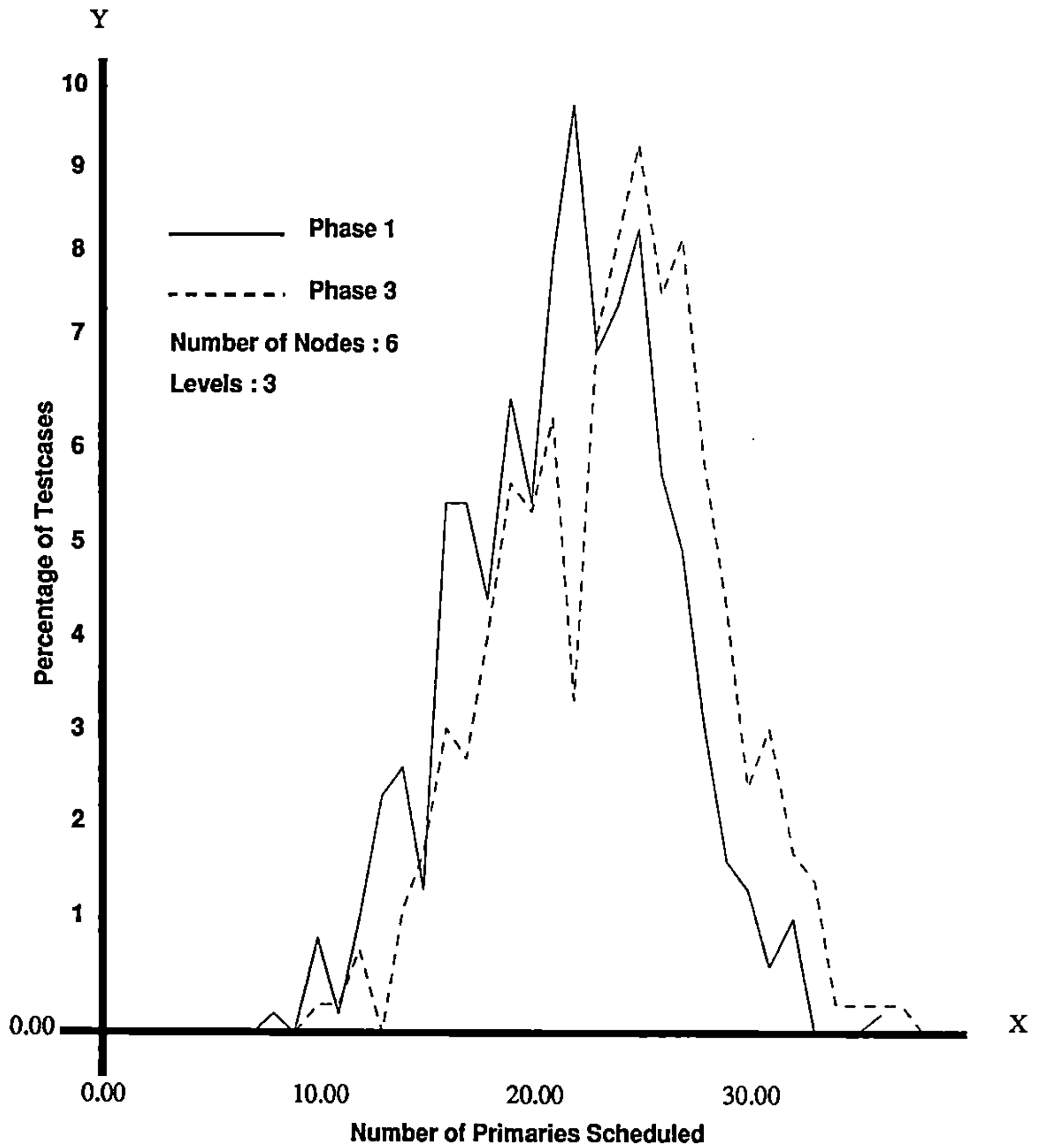


Figure 5.9: Performance of the scheduler for virtual ring of 6 nodes

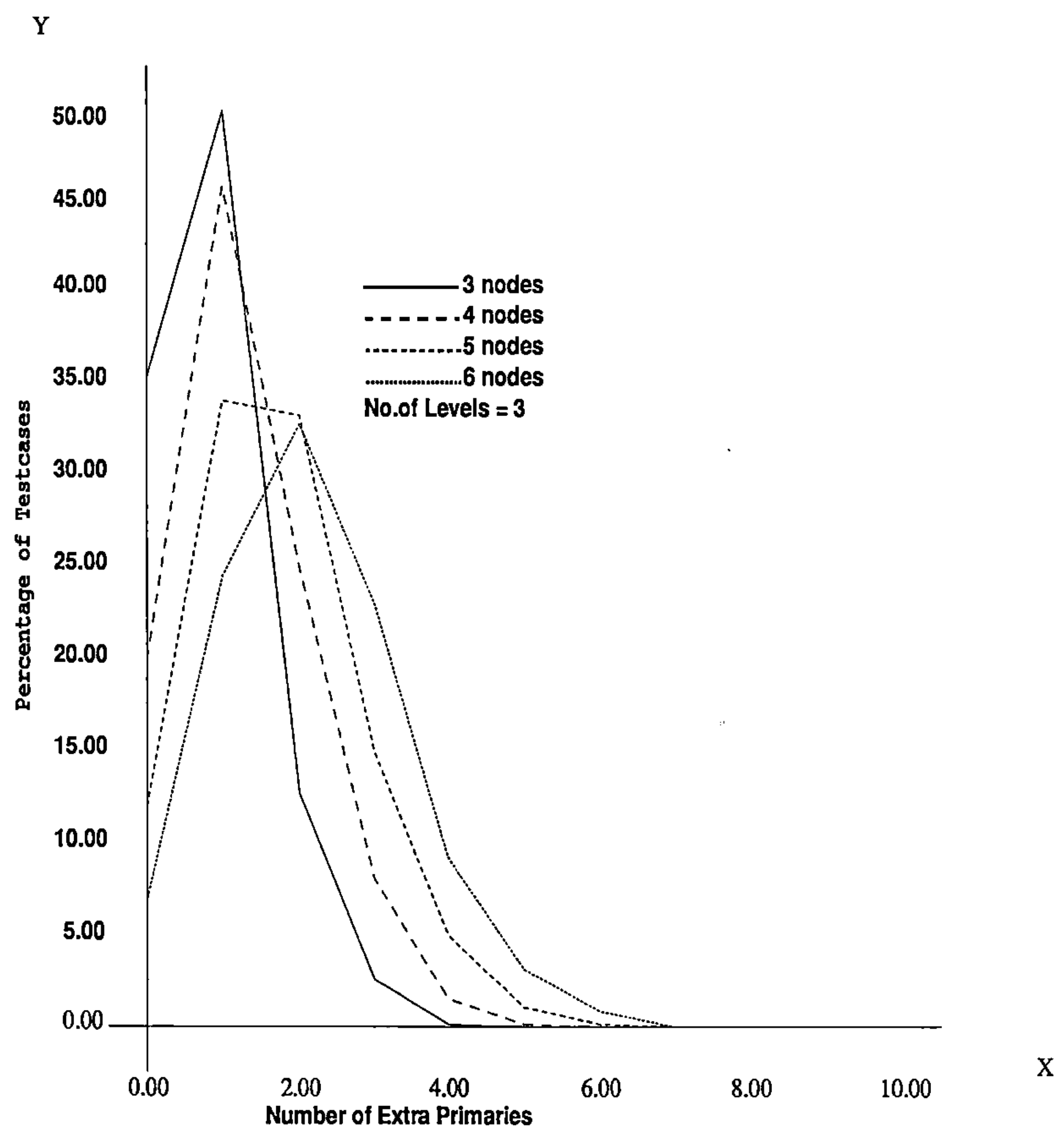


Figure 5.10: Percentage of test cases where extra primaries were scheduled for virtual ring network

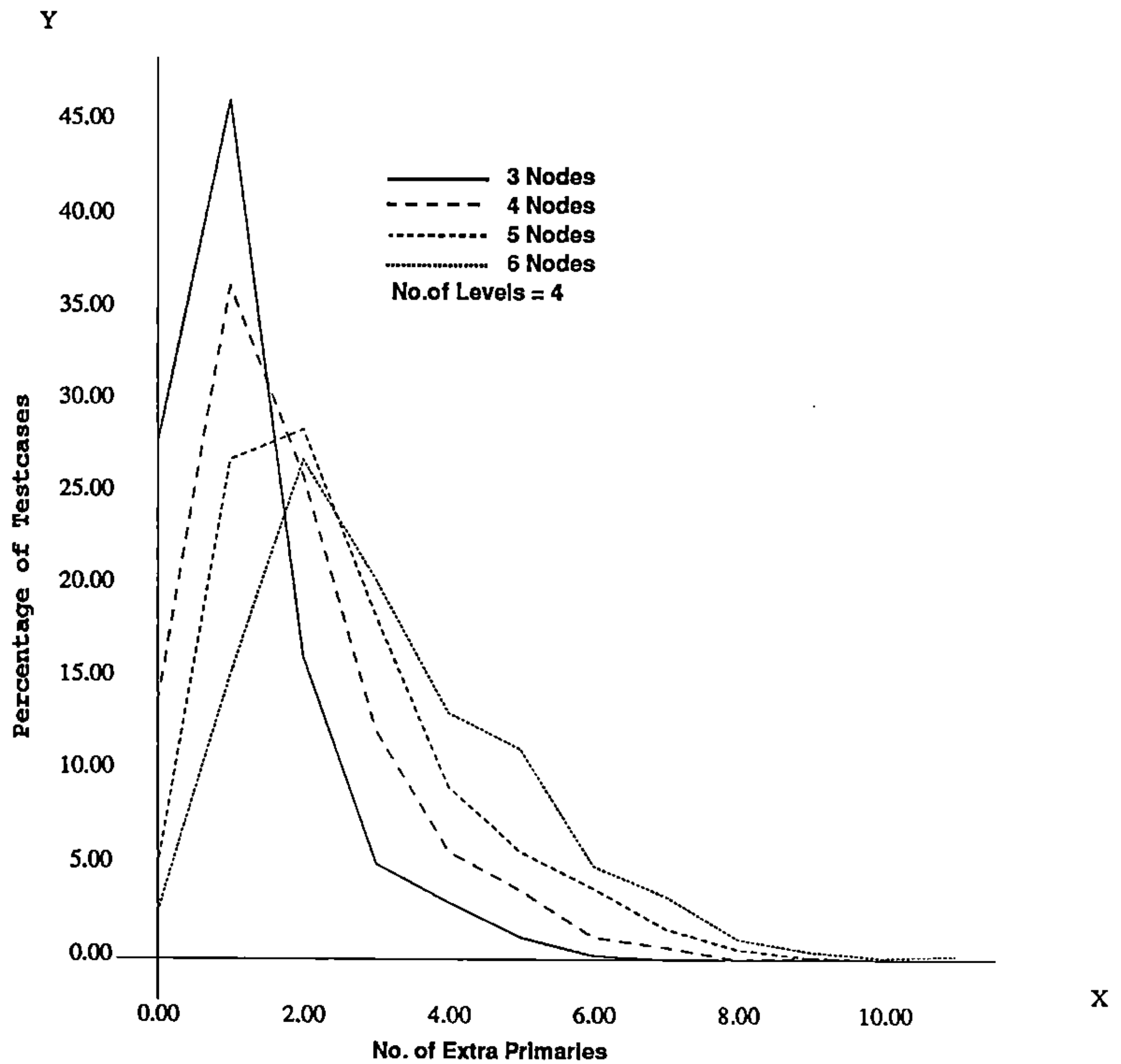


Figure 5.11: Percentage of test cases where extra primaries were scheduled for virtual ring network

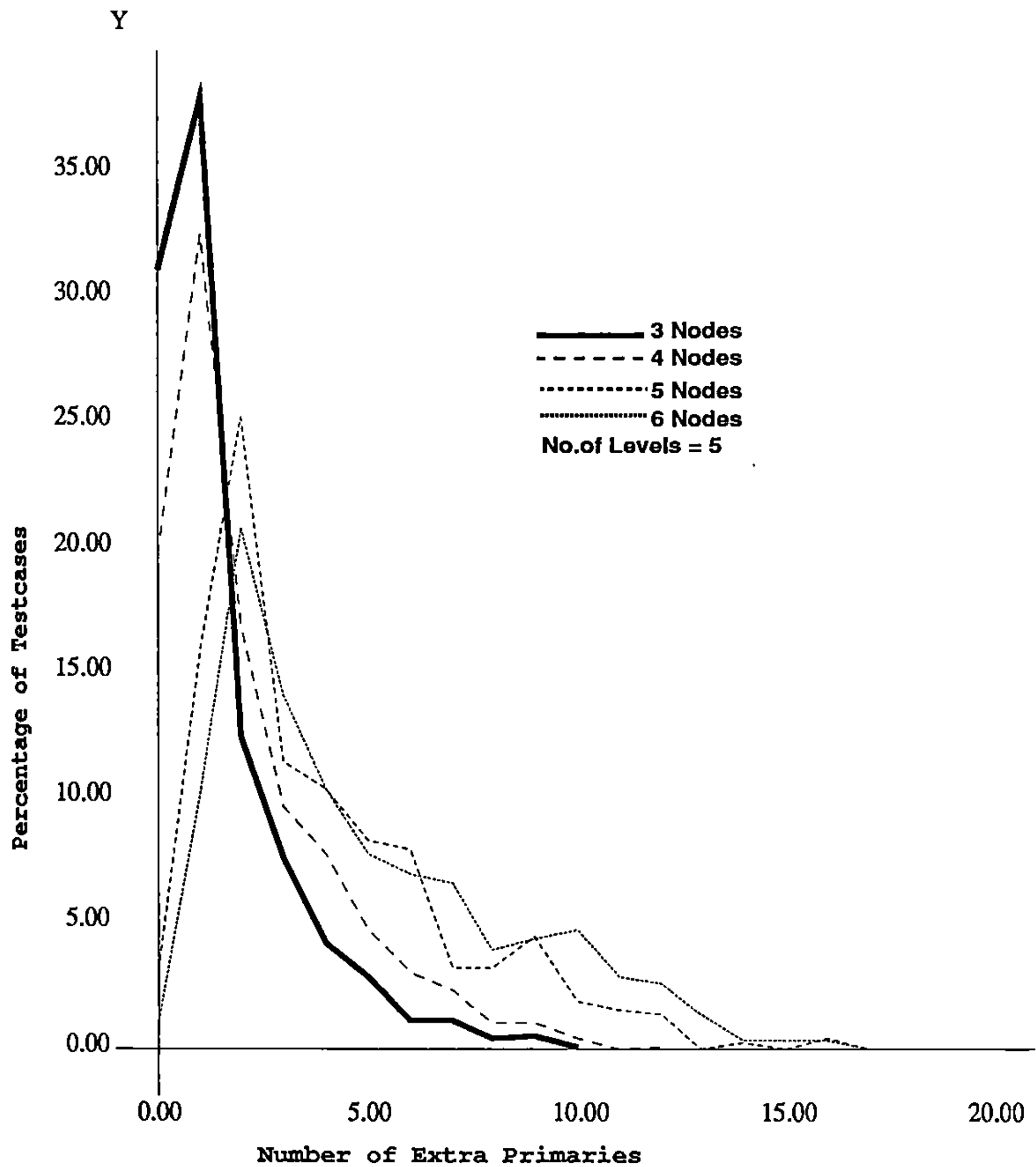


Figure 5.12: Percentage of test cases where extra primaries were scheduled for virtual ring network

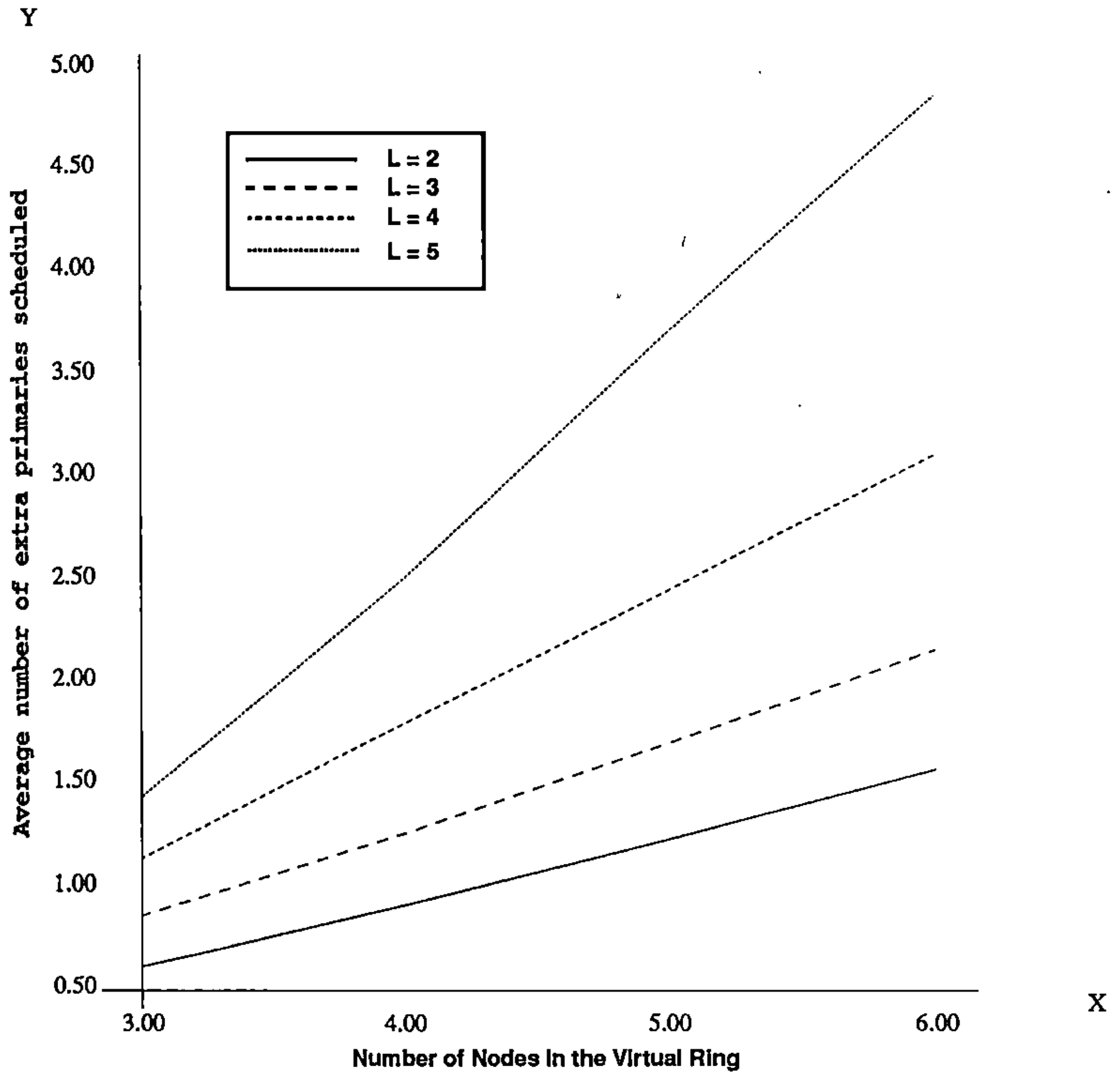


Figure 5.13: Average number of extra primaries scheduled when $T_{k+1}/T_k = 2$

5.4 Results of Simulation for Binary n-Cube

The scheduler was simulated for simply periodic Binary n-Cube interconnection networks. The execution times of the primary and alternate algorithms for various levels were generated using the random number generator. The arrival period of the lowest level, T_0 was taken to be 10 time units. Two ratios between arrival periods were chosen. In the first one, the ratio between the arrival periods for jobs of various levels, T_{l+1}/T_l was 2. The arrival periods of jobs of various levels follow the pattern of 10, 20, 40, 80,.... In the other, the ratio between the arrival periods for jobs of various levels, T_{l+1}/T_l was 3, where the arrival periods of jobs of various levels follow the pattern of 10, 30, 90, 270,.... The generated execution times of primary and alternate algorithms are in the range of $[0...T_0]$ to ensure that the jobs are schedulable. The performance of the scheduler for Binary n-Cubes of various dimensions and for various levels is discussed in the following sections.

5.4.1 Performance for Two Levels in Binary n-Cube

The performance of the scheduler was tested for Binary n-Cube of various dimensions when T_{l+1}/T_l is 2. Figure 5.14 illustrates the performance of the scheduler for a Binary n-Cube of dimension 2. Since the number of primaries to be scheduled in each individual node is 3, i.e., two $P_{0,i}$ and one $P_{1,i}$, the total number of primaries in the system is 12. The figure gives the performance of the scheduler after the Phase 1 algorithm and after the Phase 3 algorithm. It can be seen that the number of primaries scheduled by the proposed distributed scheduler after Phase 3 is greater than the number of primaries scheduled after Phase 1. Similarly, Figures 5.15 and 5.16

illustrate the performance of the scheduler after Phase 1 and after Phase 3. It is clear from the figures that as the number of dimensions of the Binary n-Cube increases, the number of additional primaries scheduled by the proposed scheduler increases. This is illustrated in Figure 5.17, where the performance of the algorithm is given in terms of the additional primaries scheduled. The drop in the percentage of testcases where no additional primaries are scheduled, as the dimensions of the Binary n-Cube increase can be noticed.

Figure 5.18 gives the performance of the scheduler when T_{l+1}/T_l is 3. The percentage of testcases where no additional primaries are scheduled for Binary 2-Cube, Binary 3-Cube and Binary 4-Cube are 42%, 15% and 3% respectively.

5.4.2 Performance for Three Levels in Binary n-Cube

The performance of our proposed scheduler when $T_{l+1}/T_l = 2$ is given. Figure 5.19 illustrates the performance of the scheduler for a Binary 2-Cube where the individual nodes have three levels. Our proposed scheduler consistently schedules additional primaries after the Phase 3 algorithm compared to the Phase 1 algorithm. It can be seen in Figures 5.20 and 5.21 that, as the number of dimensions of the Binary n-Cube increases, the number of additional primaries scheduled increases. Similarly, from Figure 5.22 it can be seen that the percentage of testcases for which no additional primaries are scheduled after Phase 3 are 40%, 4% and 1% for the Binary 2-Cube, Binary 3-Cube and Binary 4-Cube respectively.

Figure 5.23 illustrates the performance of the scheduler when the ratio T_{l+1}/T_l is 3. It is interesting to note in the graph that in all the test cases 4-Cube had scheduled extra primaries.

5.4.3 Performance for Higher Levels in Binary n-Cube

Figures 5.24 and 5.25 illustrate the additional primaries scheduled in a Binary n-Cube when the ratio of arrival period, $T_{l+1}/T_l = 2$. Similarly Figures 5.26 and 5.27 illustrate the additional primaries scheduled in a Binary n-Cube when the ratio of arrival period, $T_{l+1}/T_l = 3$.

Figures 5.28 and 5.29 show the performance of the algorithm for various levels as the dimension of the cube increases.

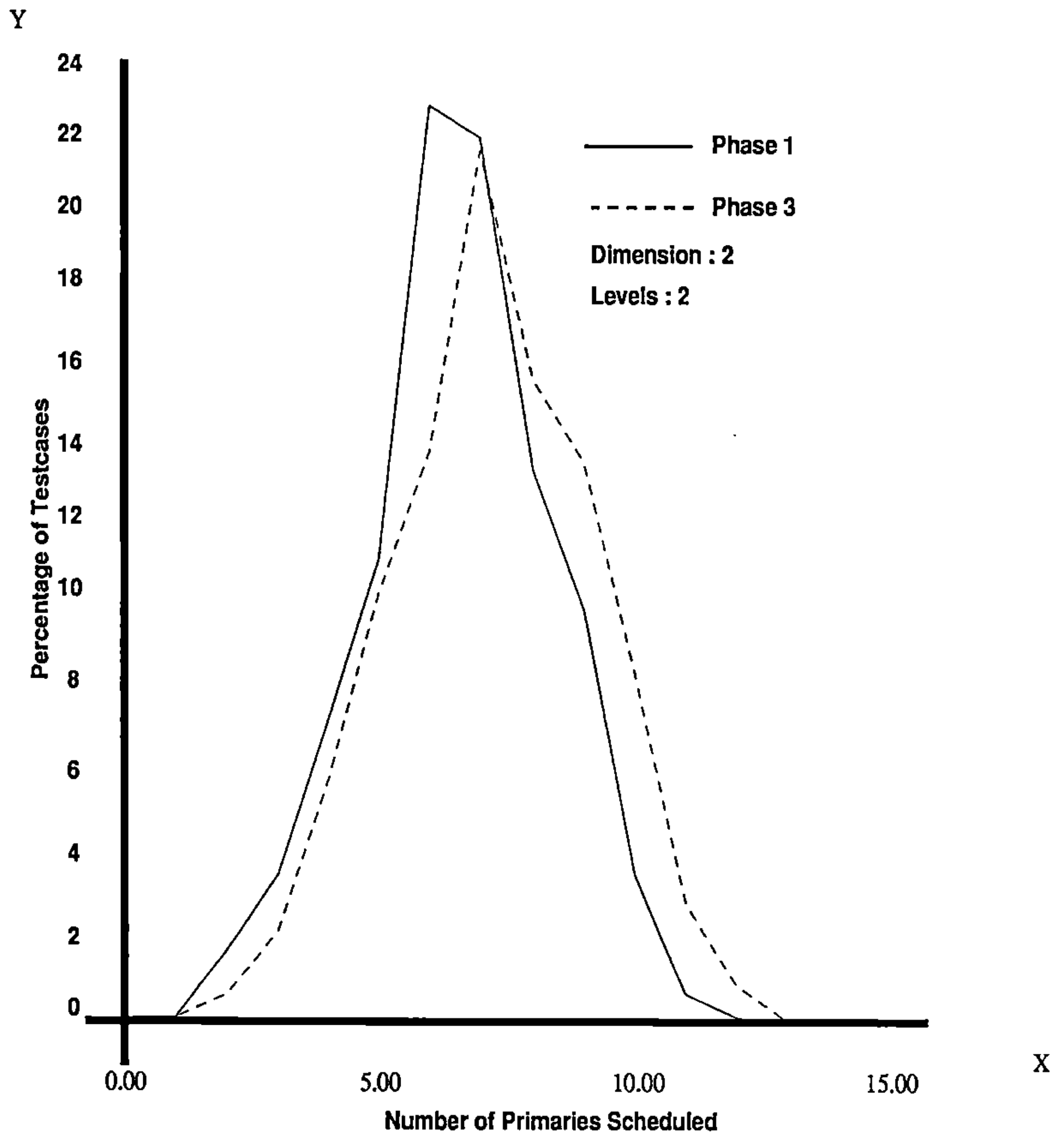


Figure 5.14: Performance of the scheduler for Binary 2-Cube

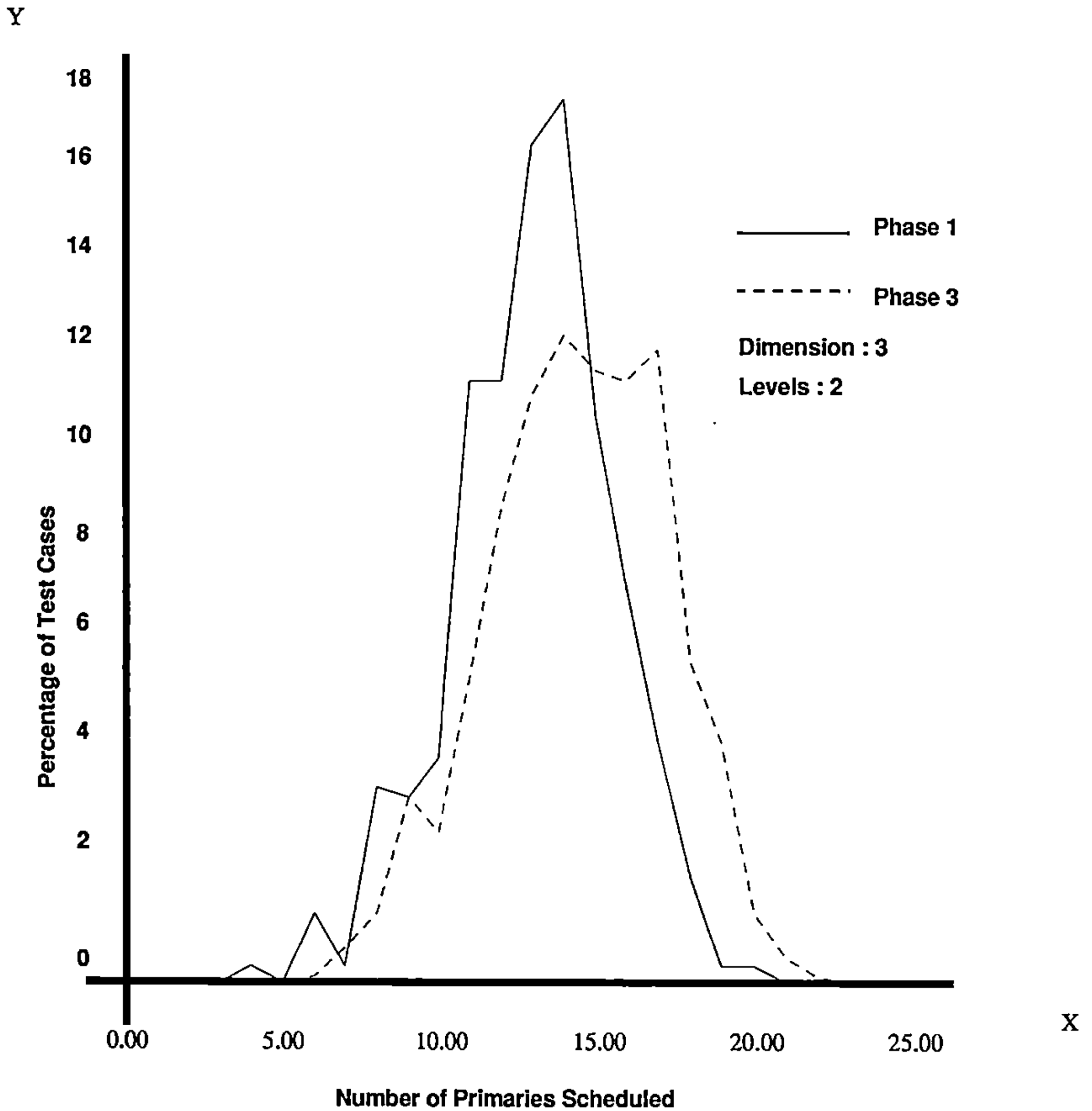


Figure 5.15: Performance of the scheduler for Binary 3-Cube

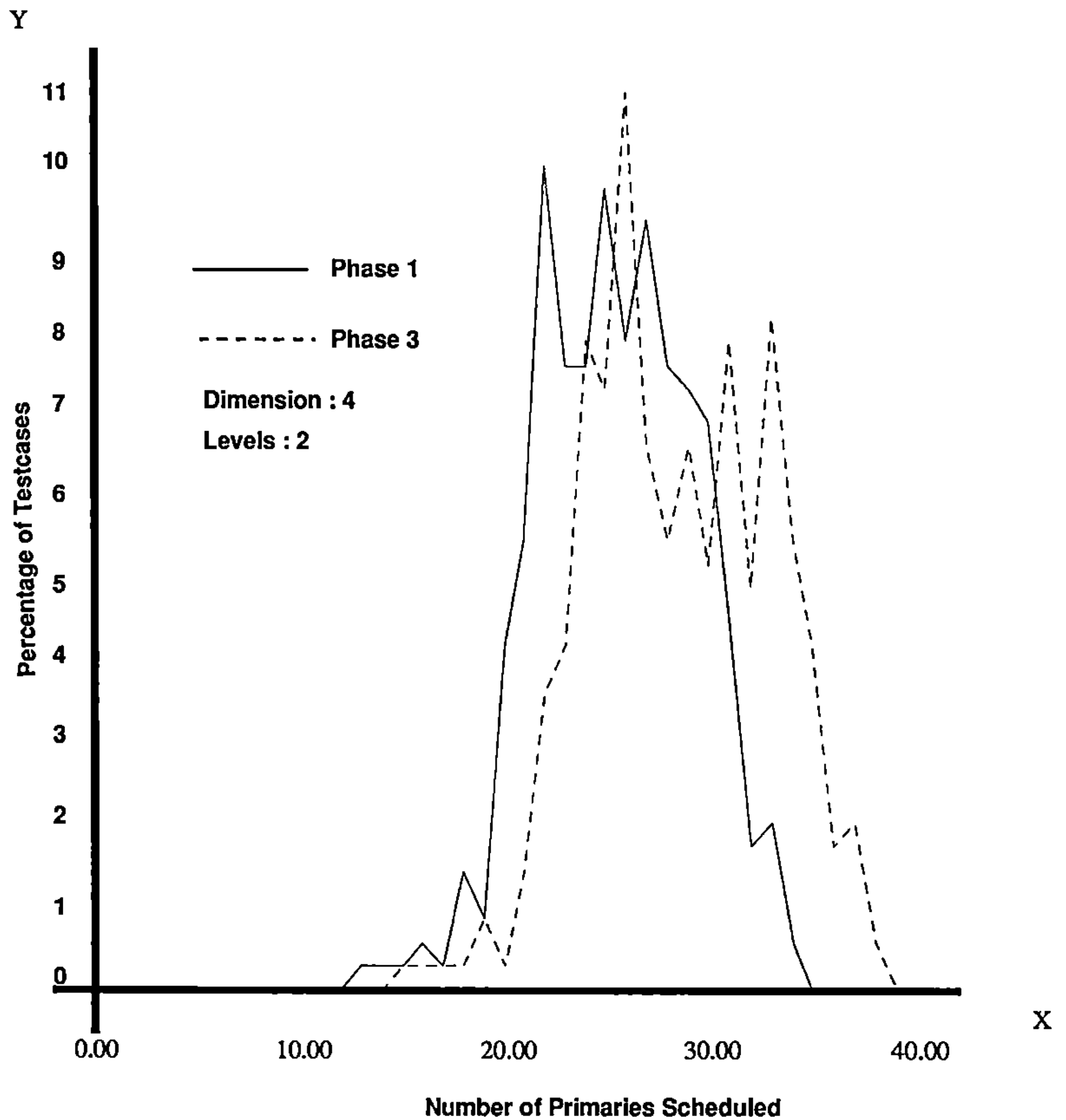


Figure 5.16: Performance of the scheduler for Binary 4-Cube

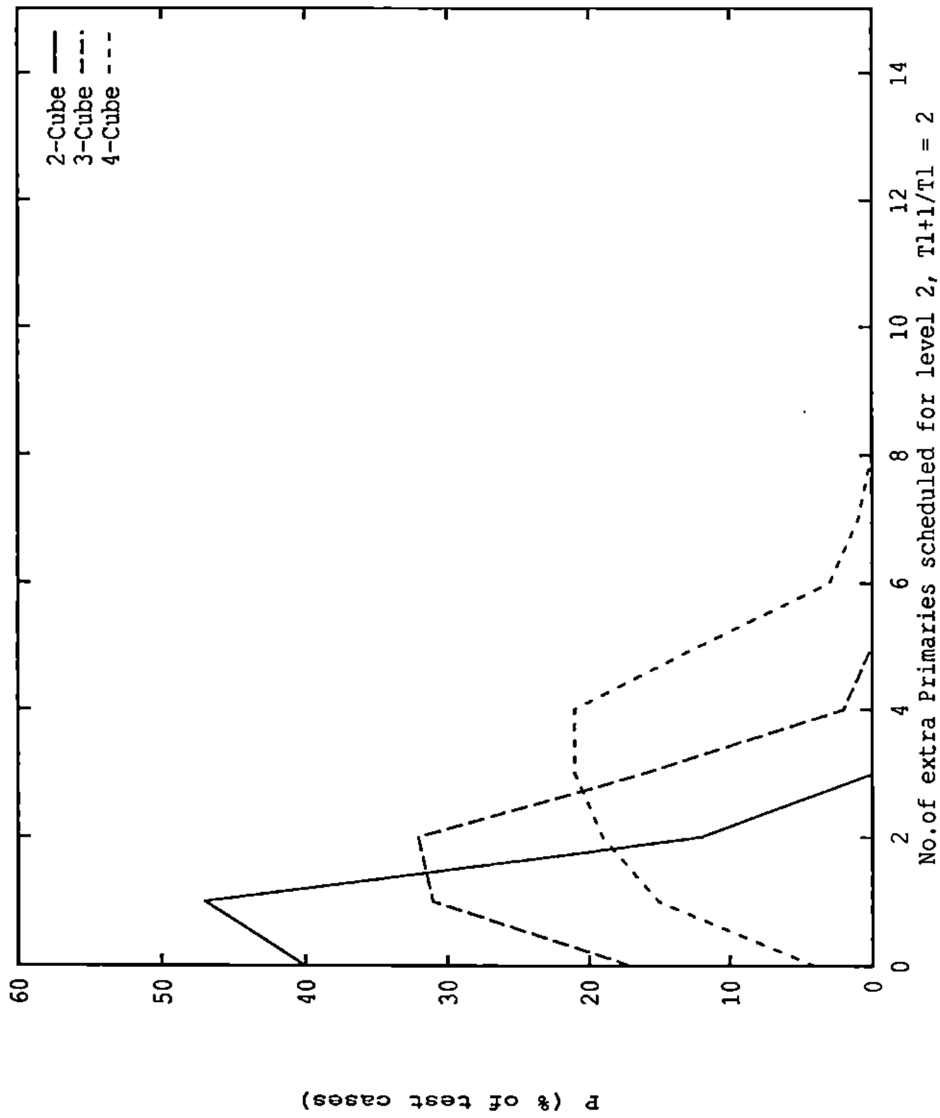


Figure 5.17: Percentage of test cases extra primaries were scheduled for n-Cube

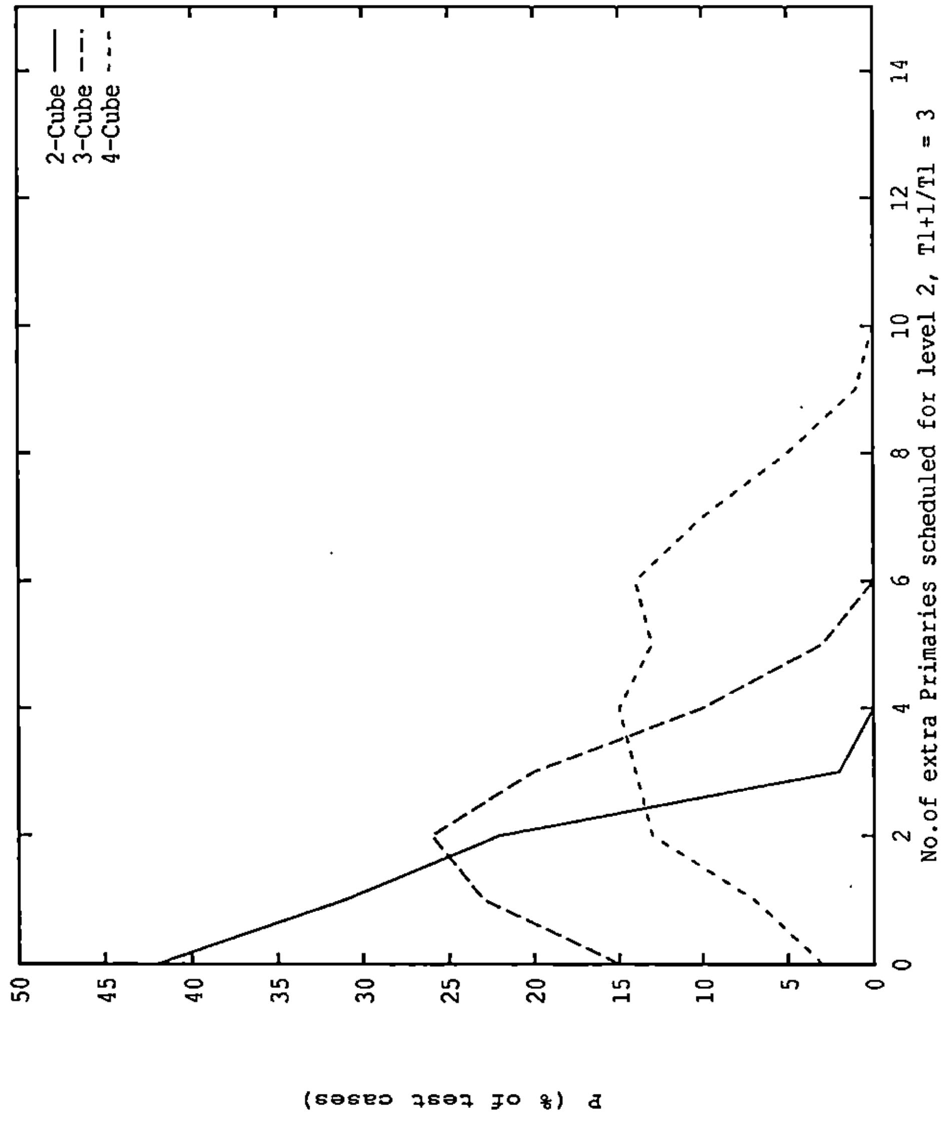


Figure 5.18: Percentage of test cases extra primaries were scheduled for n-Cube

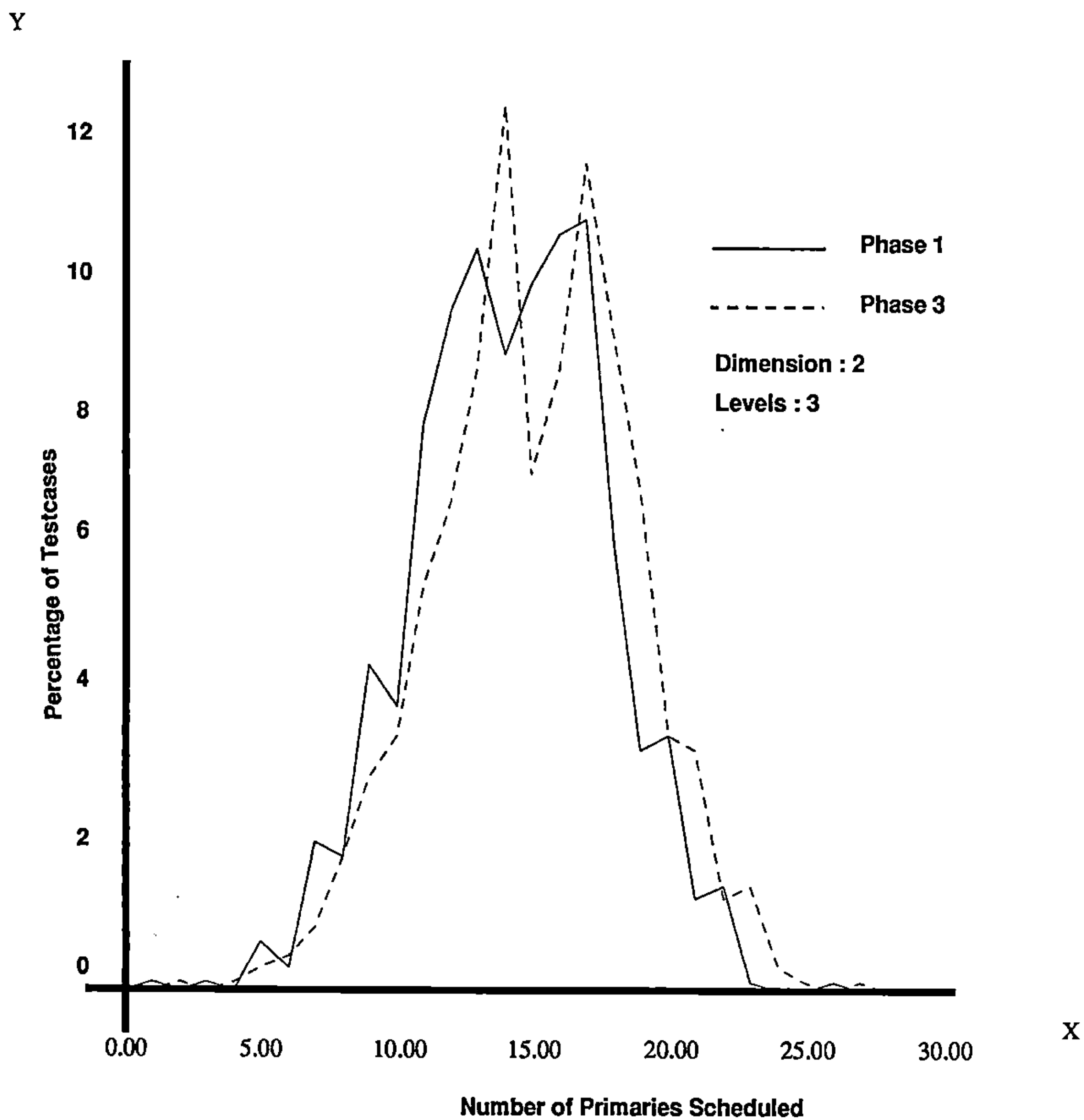


Figure 5.19: Performance of the scheduler for Binary 2-Cube

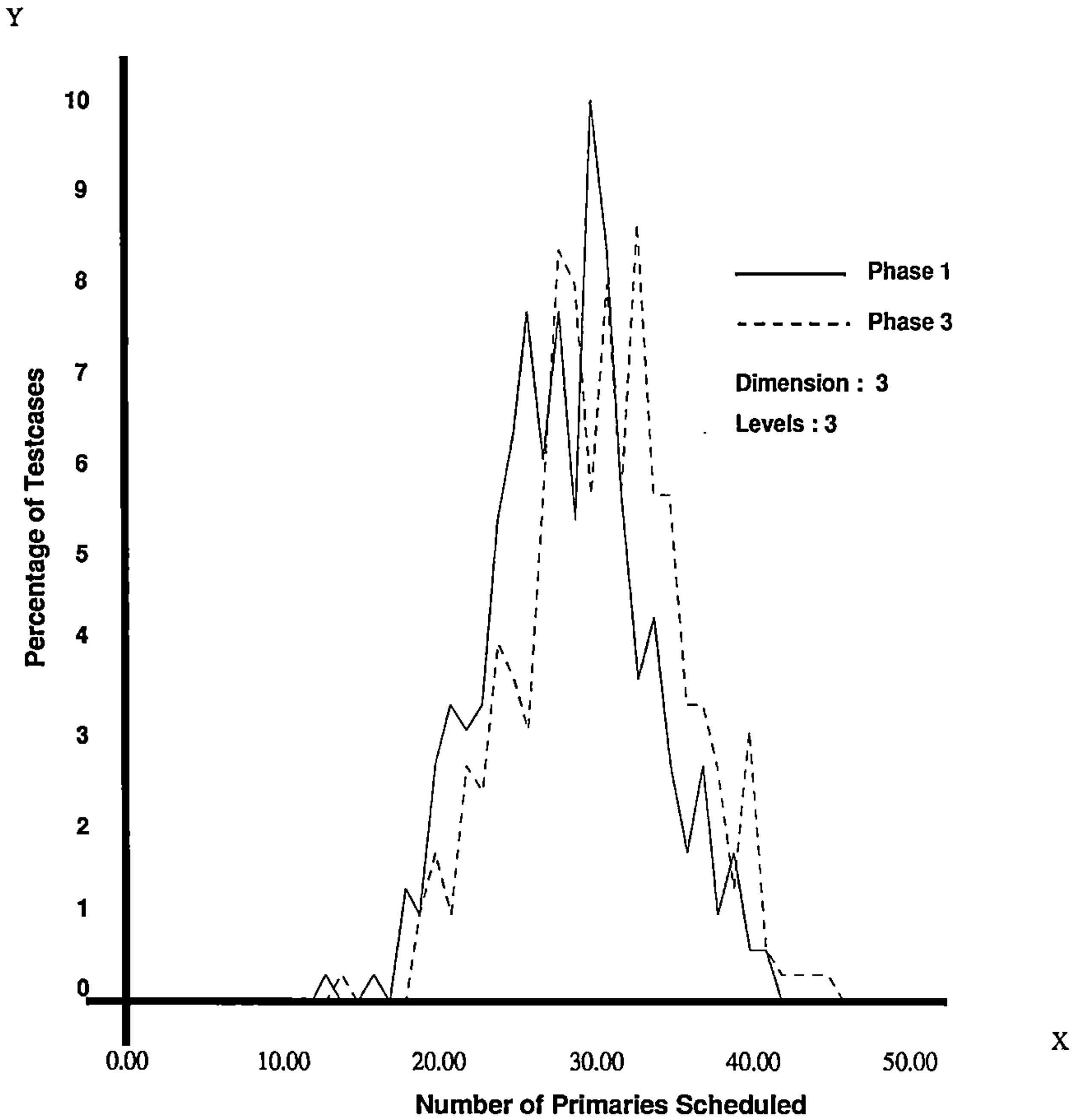


Figure 5.20: Performance of the scheduler for Binary 3-Cube

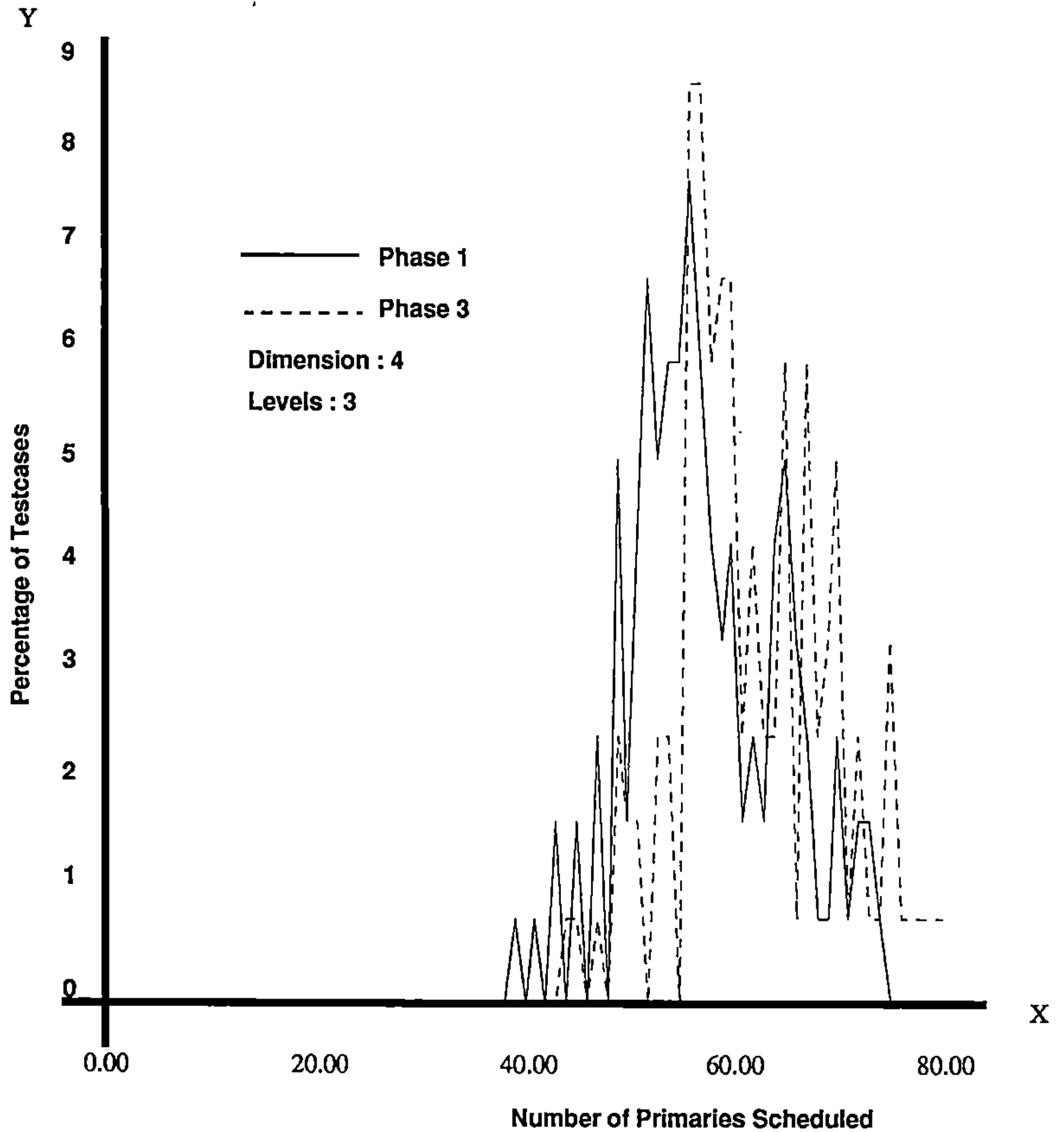


Figure 5.21: Performance of the scheduler for Binary 4-Cube

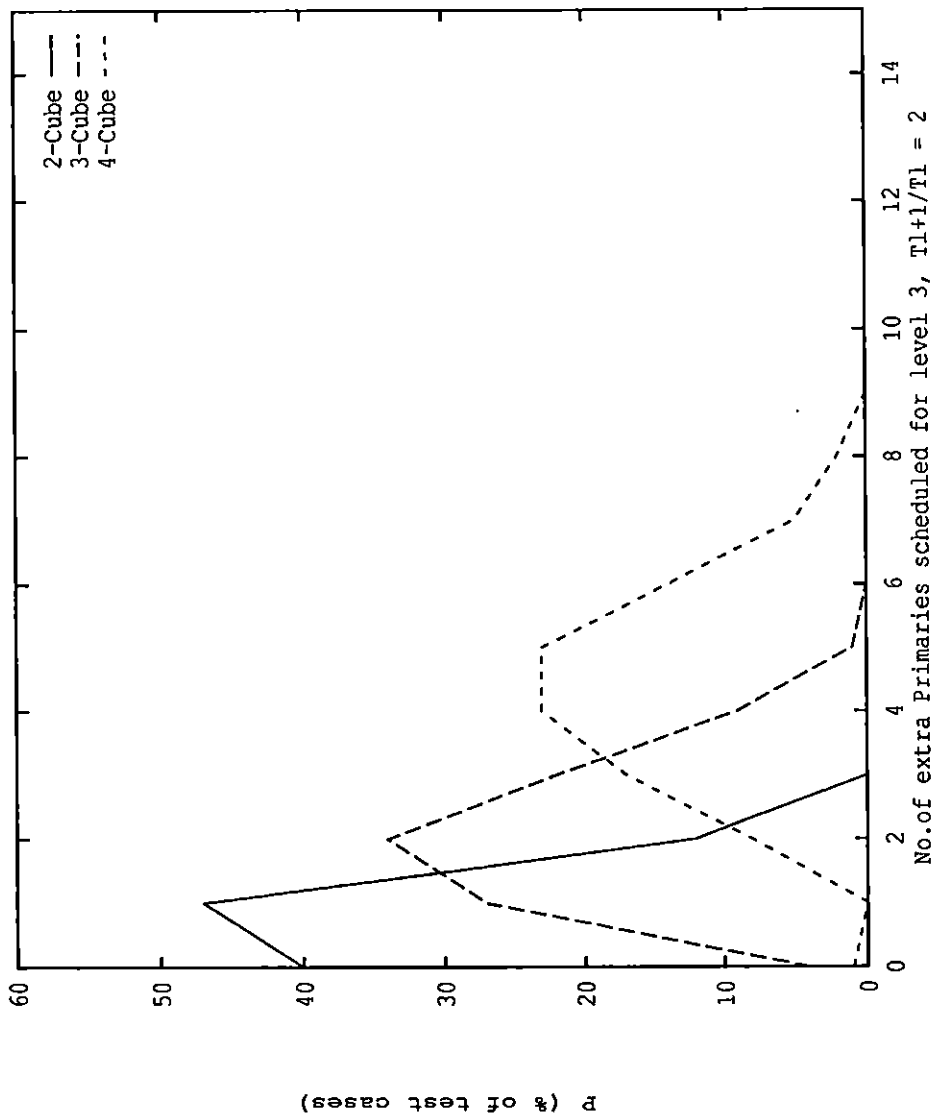


Figure 5.22: Percentage of test cases extra primaries were scheduled for n-Cube

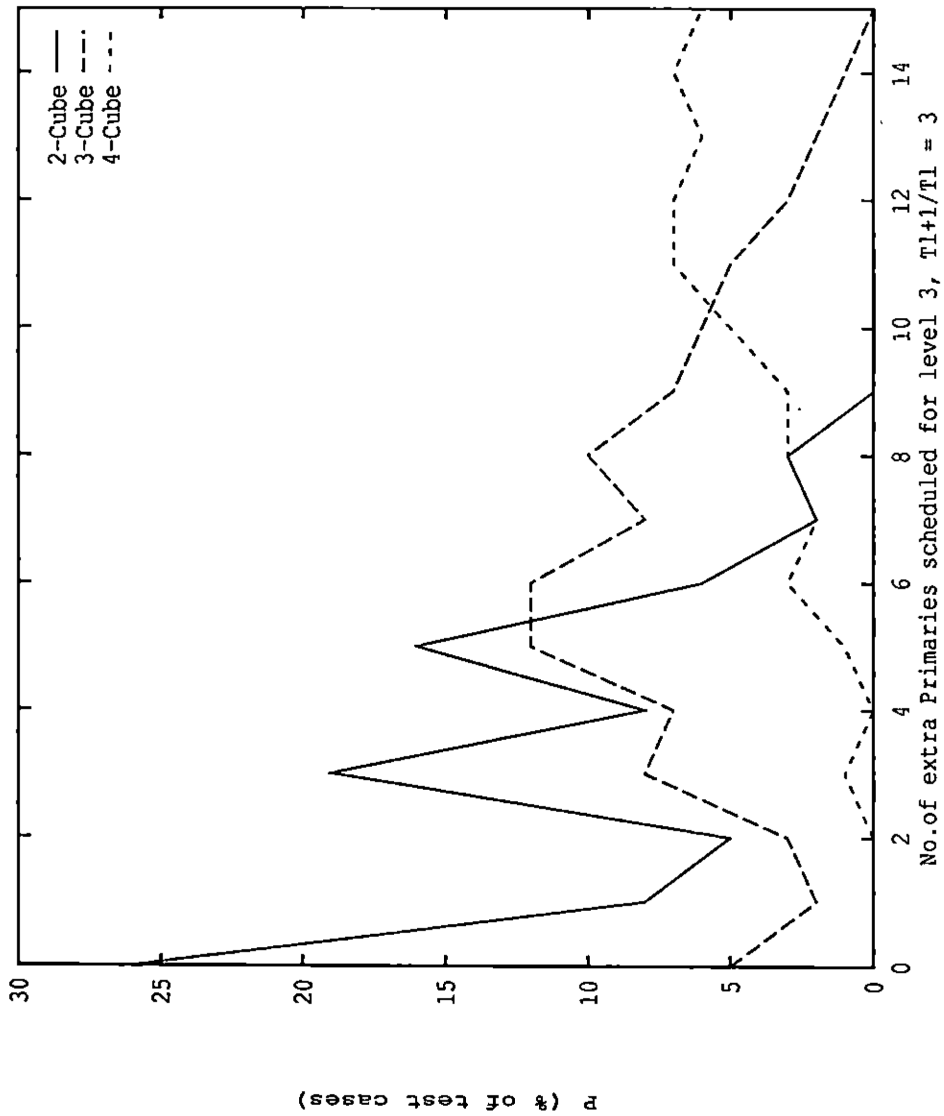


Figure 5.23: Percentage of test cases extra primaries were scheduled for n-Cube

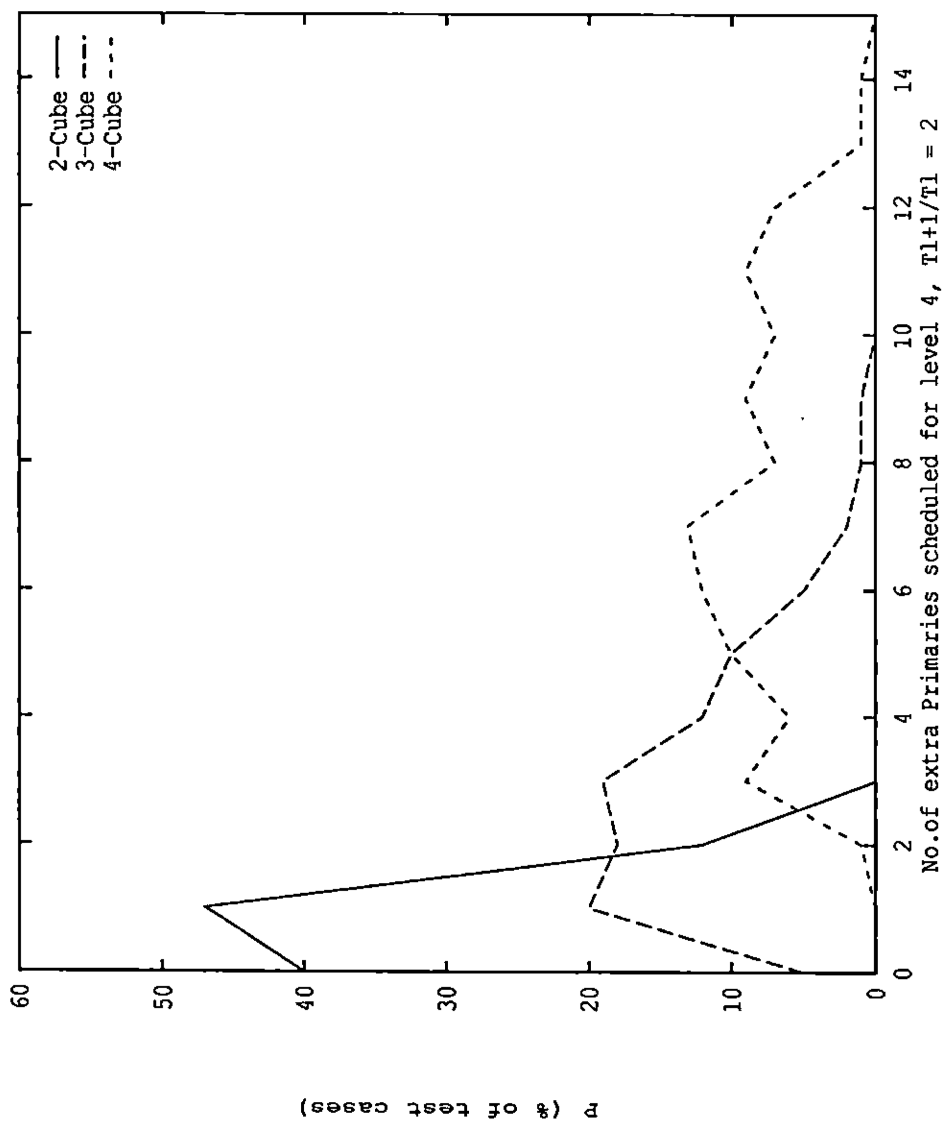


Figure 5.24: Percentage of test cases extra primaries were scheduled for n-Cube

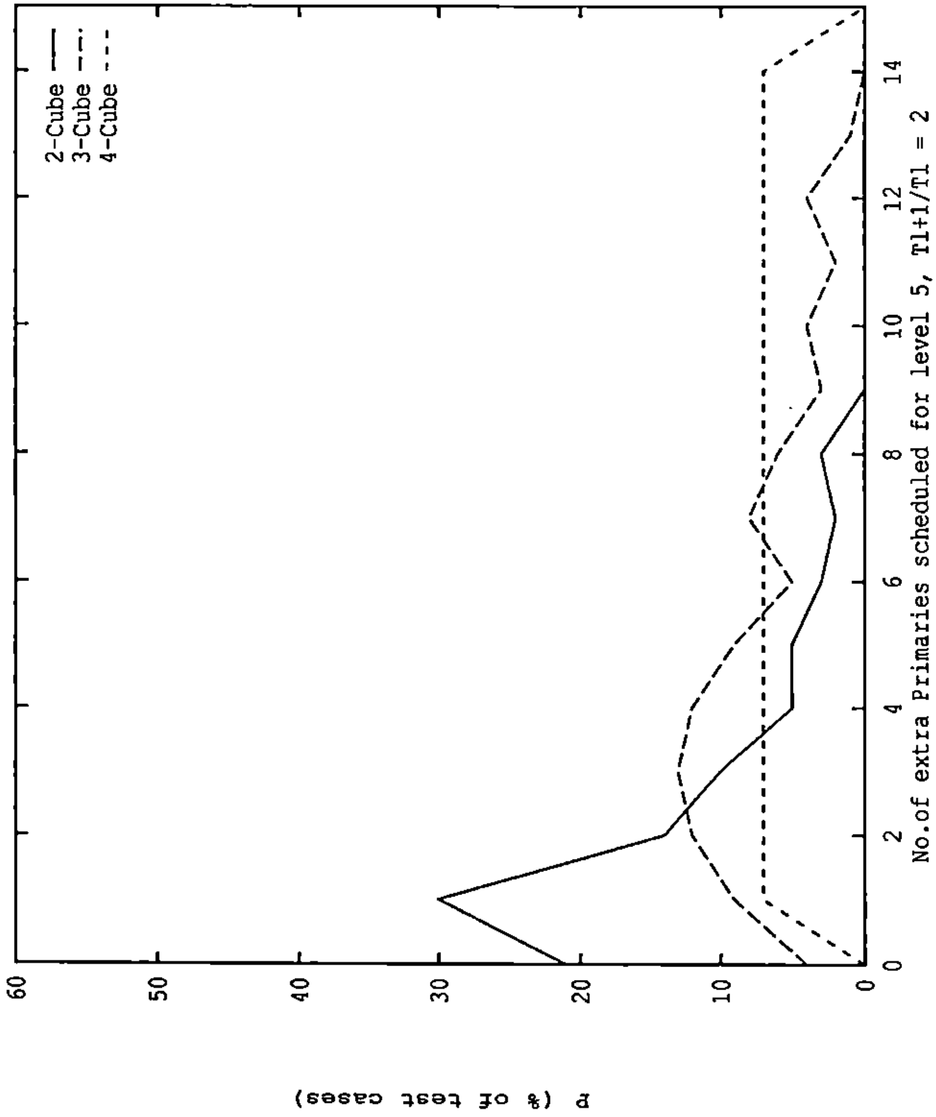


Figure 5.25: Percentage of test cases extra primaries were scheduled for n-Cube

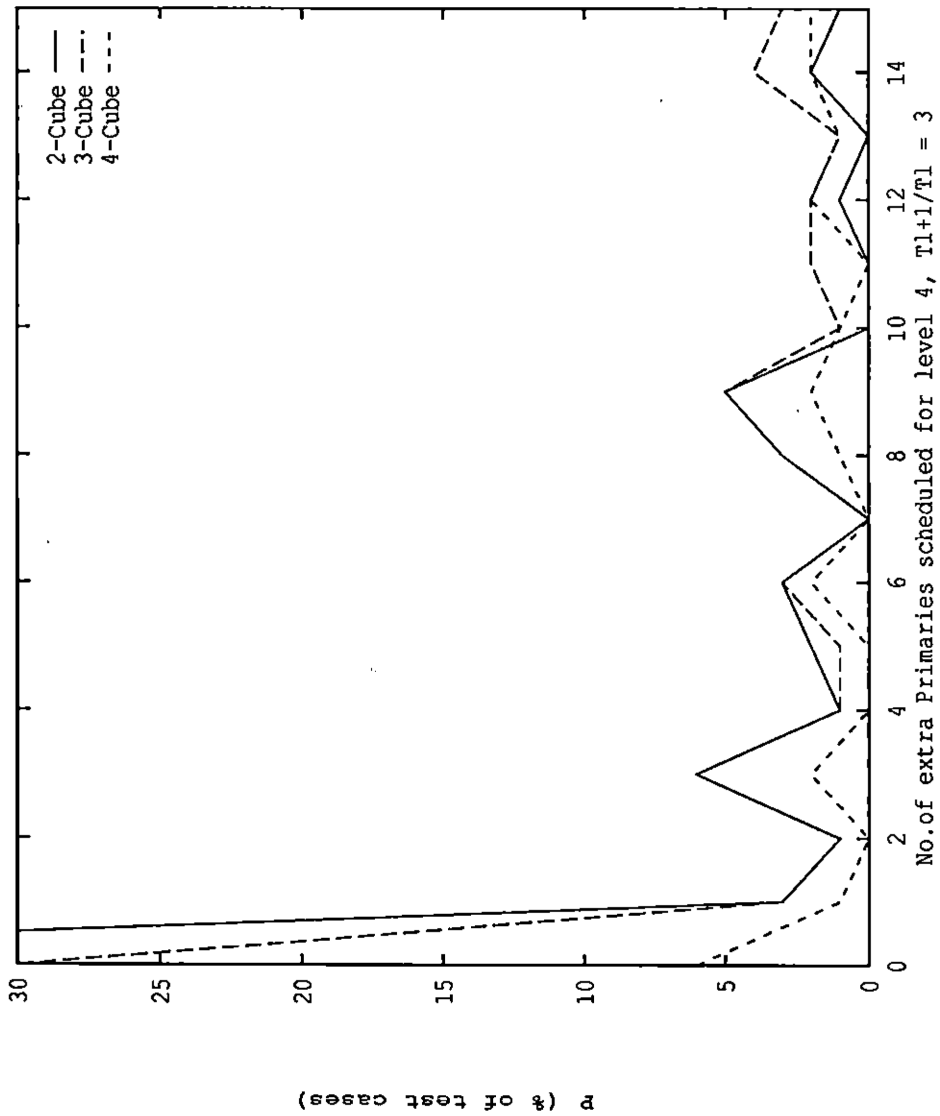


Figure 5.26: Percentage of test cases extra primaries were scheduled for n-Cube

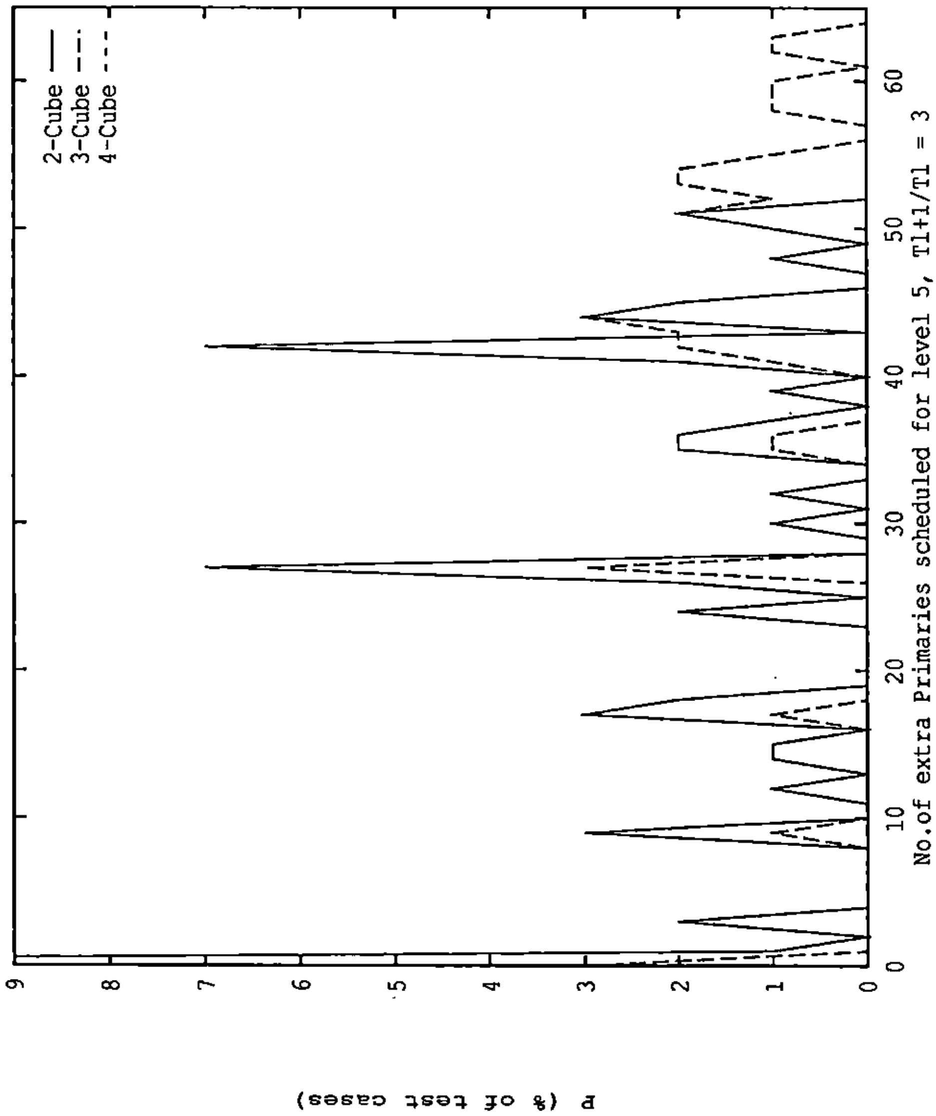


Figure 5.27: Percentage of test cases extra primaries were scheduled for n-Cube

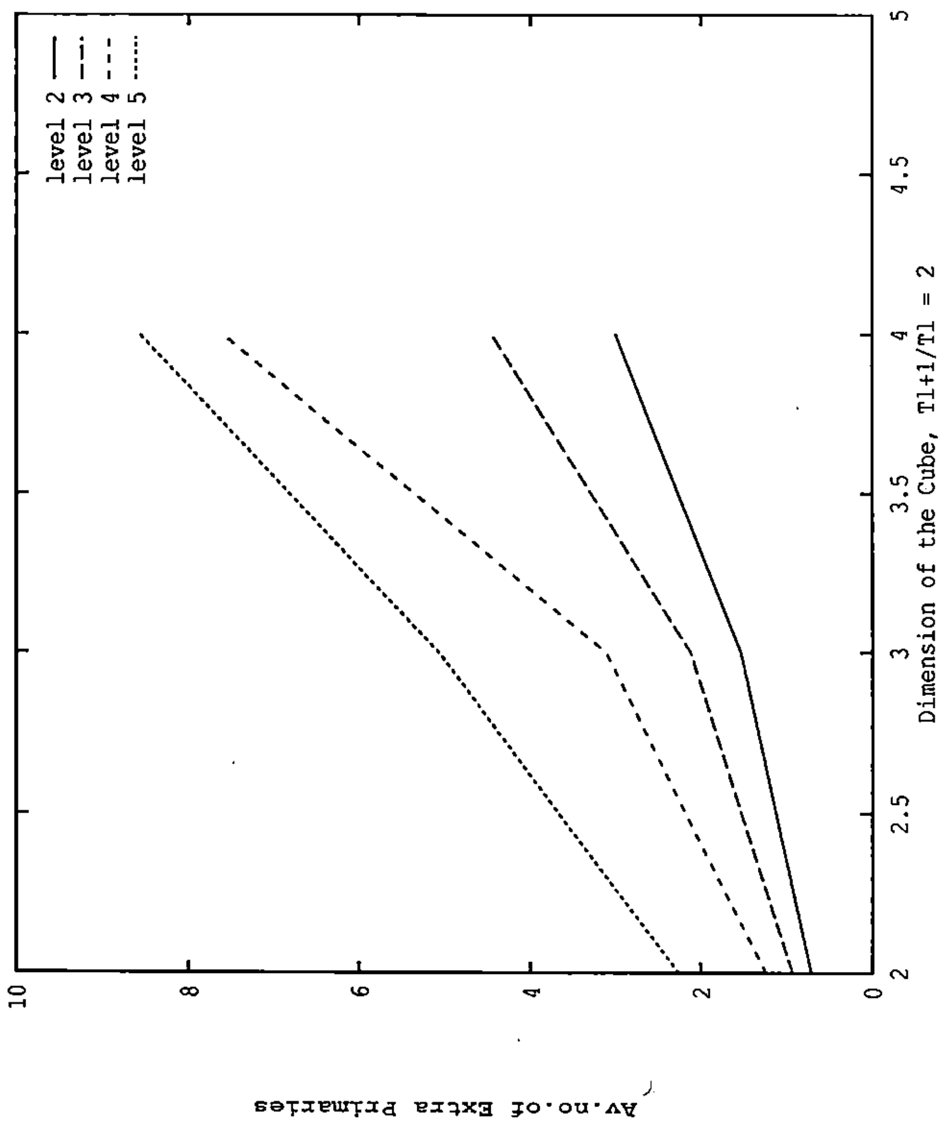


Figure 5.28: Average number of extra primaries scheduled for n-Cube

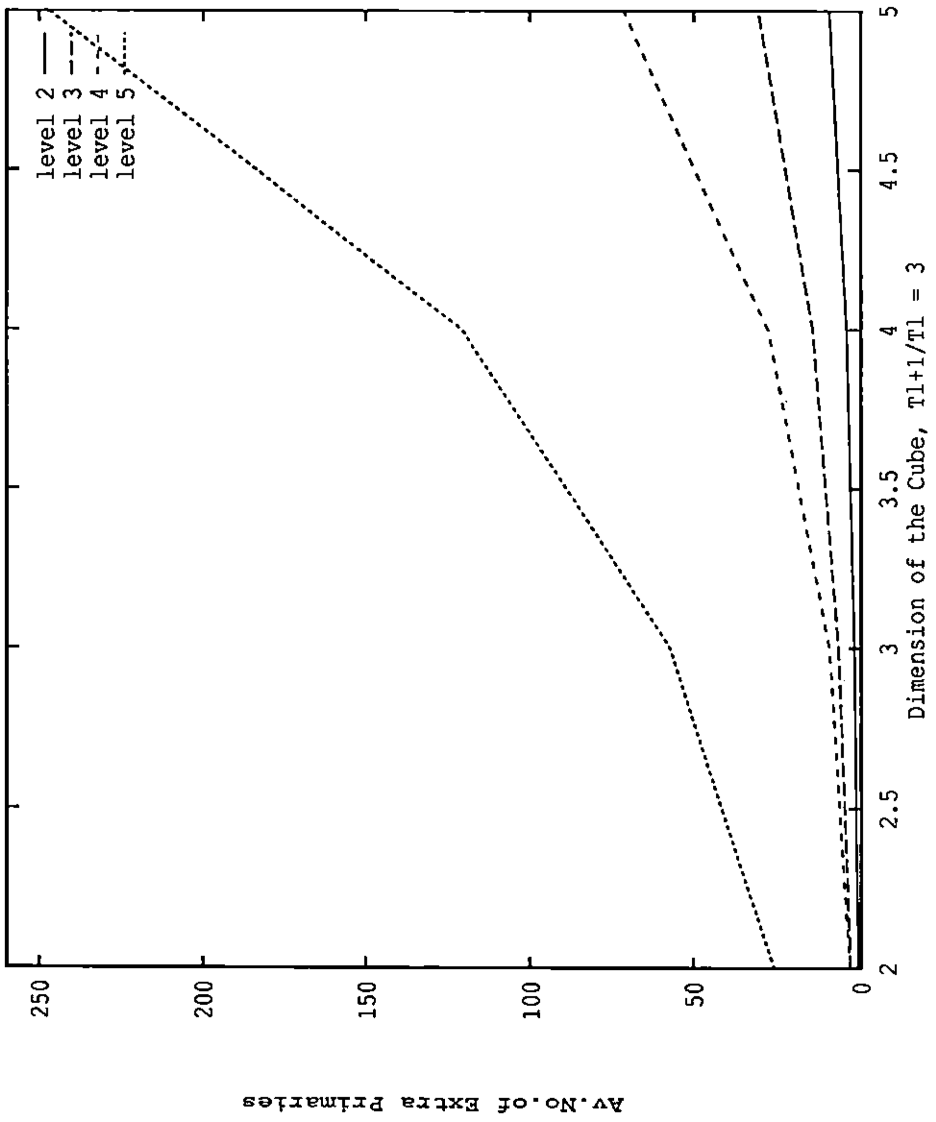


Figure 5.29: Average number of extra primaries scheduled for n-Cube

Chapter 6

Conclusions

Liestman and Campbell introduced a scheduler for single processor system for simply periodic jobs using the Deadline Mechanism. We have extended the Liestman-Campbell scheduler to the distributed environment.

A general fault-tolerant real-time scheduler for a distributed system was introduced. The scheduler was applied to two different types of networks, namely the virtual ring network and the n-cube interconnection network. The primary and the alternate algorithms of various jobs in the individual nodes of the network were scheduled using the Liestman-Campbell scheduler. In the virtual ring network, the linked list of unscheduled primaries was communicated through a cycle. In the n-cube interconnection network, the linked list of unscheduled primaries was communicated through a Hamiltonian cycle found using a Gray code technique. The unscheduled primaries of this list were scheduled in the empty slots of various nodes in the system, increasing the overall number of primaries scheduled in the system. As the primaries provide accurate result, the accuracy of result increases as the number of primaries scheduled increases. Since our distributed algorithm succeeds in scheduling more primaries, a better result accuracy is achieved.

The alternate algorithm of a job is essentially used only when a primary

fails. Thus the time slot occupied by the alternate algorithm becomes free during run-time if the corresponding primary algorithm succeeds. Therefore we have also proposed a dynamic scheduler to optimize the performance of the system, by rescheduling the unscheduled primaries in place of alternates that become redundant at run-time. The dynamic scheduler maintains a table with the information regarding all the unscheduled primary algorithms across the system. When a previously unscheduled primary algorithm is scheduled during run-time, the tables in all the nodes across the system are updated, avoiding scheduling the same primary algorithm by two different nodes.

The distributed scheduler was simulated for various randomly generated data. The results of the performance evaluation for the Virtual ring network and for the Binary n-Cube interconnection network indicate that our proposed algorithm succeeds in scheduling additional primaries as the number of nodes and the number of levels increase.

The future work includes implementation and evaluation of our proposed alternative algorithm. Also, extension of our proposed distributed algorithm for scheduling tasks with precedence constraints with periodic tasks will be of special interest to many process control applications as well as applications such as the robot arm manipulator problem.

Bibliography

[1] J. Stankovic, "Real-Time Computing Systems : The Next Generation," in *Tutorial : Hard Real-Time Systems* . IEEE Computer Society, pp. 14–37, 1988.

[2] J. D. Schoeffler, "Distributed Computer Systems for Industrial Process Control," *IEEE Computer*, 17, no. 2, pp. 11–18, February 1984.

[3] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak and C. B. Weinstock, "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proceedings of the IEEE*, 66, no. 11, pp. 1240–1255, October 1978.

[4] G. D. Carlow, "Architecture of the Space Shuttle Primary Avionics Software System," *Communications of ACM*, 27, no. 9, pp. 926–936, September 1984.

[5] H. Kasahara and S. Narita, "Parallel Processing of Robot-Arm Control Computation on a MultimicroprocessorSystem," *IEEE Journal of Robotics and Automation*, RA-1, no. 2, pp. 3–6, June 1985.

[6] N. Wirth, "Toward a discipline of real-time programming," *Communication of ACM*, 20, no. 8, pp. 577–583, August 1977.

[7] B. Dasarathy, "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them," *IEEE Transactions on Software Engineering*, 11, no. 1, pp. 80–86, January 1985.

[8] F. Jahanian and A. K. Mok, "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Transactions on Software Engineering*, 12, no. 9, pp. 890–904, September 1986.

[9] W. A. Horn, "Some simple scheduling algorithms," *Naval Research Logistics Quarterly*, 21, 1974.

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

[11] J. P. Lehoczky and L. Sha, "Performance of bus scheduling algorithms," in *Performance 86.* pp. 44–53, 1986.

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

[13] S. Davari and S. K. Dhall, "An on line algorithm for real-time tasks allocation," in *IEEE Proceedings on Real-Time System Symposium.* pp. 194–200, December 1986.

[14] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, 26, no. 1, pp. 127–140, 1978.

[15] J. K. Lenstra, A. H. G. R. Kan and P. Bruchker, "Complexity of machine scheduling problems," in *Annals of Discrete Mathematics, North Holland Publishers* , no. 1. 1977.

[16] J. D. Ullman, "NP-Complete Scheduling Problem," *Journal of Computer and System Sciences*, 10, pp. 384–393, 1975.

[17] B. Simons, "A fast algorithm for multiprocessor scheduling," in *Proceedings 21st annual symposium on Foundation of Computer Science*, vol. 21. 1980.

[18] B. Simons, "Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines," *SIAM Journal for Computing*, 12, no. 2, 1983.

[19] B. Simons and M. Sipser, "On scheduling unit-length jobs with multiple release time or deadline intervals," *Operations Research*, 32, no. 1, pp. 80–88, 1984.

[20] E. L. Lawler, "Optimal scheduling of a single machine subject to precedence constraints," *Management Science*, 19, 1973.

[21] J. Blazewicz, M. Drabowski and J. Weglarz, "Scheduling multiprocessor tasks to minimize schedule length," *IEEE Transactions on Computer*, C-35, no. 5, pp. 389–393, 1986.

[22] G. K. Manacher, "Production and stabilization of real-time task schedules," *Journal of ACM*, 14, no. 3, pp. 439–465, 1967.

[23] J. D. Ullman, "Complexity of Sequence Problems," in *Computer and Job-Shop Scheduling Theory* E.G. Coffman, editor . New York, NY: John Wiley & Sons, 1976.

[24] K. R. Baker and Z. -S. Su, "Sequencing with due-dates and early start times to minimize maximum tardiness," *Naval Research Logistics Quarterly*, 21, 1974.

[25] H. Stone, "Multiprocessor scheduling with the aid of network flow algorithms ," *IEEE transactions on Software Engineering*, SE-3 , no. 1, pp. 85–93, 1977.

[26] H. Stone, "Program assignment in three-processor systems and tricutset partitioning of graphs," University of Massachusetts, Amherst, Tech. Report ECE-CS-77-7, 1977.

[27] V. M. Lo, "Heuristic algorithms for task assignment in distributed systems," in *Proceedings of International Conference on Distributed Computing Systems.* pp. 30–39, 1984.

[28] K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *Computer*, 15, pp. 50–56, July 1982.

[29] R. P. Ma, E. Lee and M. Tsuchiya, "A Task Allocation Model for Distributed Computing Systems," *IEEE Transactions on Computers*, 31, no. 1, pp. 41–47, January 1982.

[30] D. W. Leinbaugh and M. R. Yamini, "Guaranteed Response Times in a Distributed Hard-Real-Time Environment," *IEEE Transactions on Software Engineering*, 12, no. 12, pp. 1139–1144, December 1986.

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

[32] A. K. Mok and M. L. Dertouzos, "Multiprocessor Scheduling in a Hard Real-Time Environment," in *Proceedings of the Seventh Texas Conference on Computing Systems.* Houston, Texas: pp. 5.1 – 5.12, November 1978.

[33] M. Dertouzos, "Control Robotics : the procedural control of physical processes," in *Proceedings of the IFIP Congress.* pp. 807–813, 1974.

[34] K. Ramamritham and J. Stankovic, "Dynamic task scheduling in distributed hard real-time systems," *IEEE Software*, 1, no. 3, pp. 65–75, July 1984.

[35] A. K. Mok and M. L. Dertouzos, "Multiprocessor On-Line Scheduling of Hard Real-Time Tasks," *IEEE Transactions on Software Engineering*, 15, no. 12, pp. 1497 – 1506, December 1989.

[36] E. D. Jensen, D. Locke and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Operating System," in *IEEE Proceedings of Real-Time System Symposium*, vol. 6. pp. 112–122, 1985.

[37] W. Zhao, K. Ramamritham and J. Stankovic, "Preemptive Scheduling under Time and Resource Constraints," *IEEE transactions on Computers*, 36, no. 8, pp. 225–236, August 1987.

[38] W. Zhao, K. Ramamritham and J. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems," *IEEE transactions on Software Engineering*, 13, no. 5, pp. 564–577, May 1987.

[39] L. M. Casey, "Decentralized Scheduling," *The Australian Computer Journal*, 13, no. 2, pp. 58–63, May 1981.

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

[41] M. Livny and M. Melman, "Load balancing in homogeneous broadcast distributed systems," in *Proceedings ACM Computer Network Performance Symposium.* 1982.

[42] J. Stankovic, "Simulation of three adaptive, decentralized controlled, job scheduling algorithms," *Computer Network*, 8, no. 3, 1984.

[43] J. Stankovic, "An application of bayesian decision theory to decentralized control of job scheduling," *IEEE Transactions on Computer*, C-34, no. 2, pp. 117–130, 1985.

[44] J. Stankovic, "Stability and distributed scheduling algorithms," *IEEE Transactions on Software Engineering*, SE-11, no. 10, pp. 1141–1152, 1985.

[45] J. Stankovic and R. Mirchandaney, "Using stochastic learning automata for job scheduling in distributed processing systems," in *Journal of Parallel and Distributed Computing.* pp. 527–552, December 1986.

[46] J. Stankovic, K. Ramamritham and S. Cheng, "Evaluation of a Flexible Task Scheduling Algorithm for Hard Real-Time Systems," *IEEE transactions on Computers*, 34, no. 12, pp. 1130–1143, August 1987.

[47] S. Cheng, J. Stankovic and K. Ramamritham, "Dynamic Scheduling of groups of tasks with precedence constraints in distributed hard real-time systems," in *IEEE Proceedings on Real-Time Systems Symposium*, vol. 7. pp. 166–174, May 1986.

[48] R. Butler, "An analysis of the real-time application capabilities of the SIFT Computer System," NASA, Tech. Memo. TM-84482, April 1982.

[49] J. Goldberg and et al., "Development and Analysis of the Software Implemented Fault-Tolerance (SIFT) Computer," presented at Final Report NASA Contract NASA CR-172146, February 1984.

[50] C. J. Walter, R. M. Kieckhafer and A. M. Finn, "MAFT: A Multicomputer Architecture for Fault-Tolerance in Real-Time Control Systems," in *IEEE Proceedings of the Real-Time Systems Symposium*, vol. 6 . pp. 133–140, December 1985.

[51] D. Palumbo, D. Palumbo and R. Butler, "SIFT - A Preliminary Evaluation," in *Proceedings of Digital Avionics Systems Conference.* Seattle, WA: pp. 21.4.1–21.4.6, October 1983.

[52] S. J. Larimer and S. L. Maher, "Reconfiguring Multi-Microprocessor Flight Control System," Air Force Wright Aeronautical Laboratories, Tech. Report AFWAL-TR-81-3070, May 1981.

[53] A. Hopkins, "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," *Proceedings of IEEE*, 66, no. 10, pp. 1221–1239, October 1978.

[54] A. Whiteside and et al., "Fault-Tolerant Multicomputer System for Control Applications," in *11th IEEE Conference Proceedings on Fault-Tolerant Computing*, vol. 11. Portland: p. 286, June 1981.

[55] J. J. Horning, "A program structure for error detection and recovery," in *Lecture Notes in Computer Science 16.* New York: Springer-Verlag, pp. 171–187, 1974.

[56] B. Randell, "System Structure for Software Fault-Tolerance," in *Proceedings, International Conference on Reliable Software.* pp. 437–439, 1975.

[57] H. Hecht, "Fault-Tolerant Software for real-time applications," *ACM Computing Surveys*, 8, no. 4, pp. 391–406, December 1976.

[58] L. Chen and A. Avizienis, "N-Version Programming : A Fault-Tolerance approach to reliability of Software operation," *8th annual international conference on Fault-Tolerant Computing*, 8, pp. 3–9, June 1978.

[59] R. H. Campbell, K. H. Horton and G. C. Belford, "Simulations of a Fault-Tolerant Deadline Mechanism," *9th IEEE International Conference on Fault-Tolerant Computing*, 9, pp. 95–101, 1979.

[60] T. Anderson and J. C. Knight, "A framework for software fault tolerance in real-time systems," *IEEE Transactions on Software Engineering*, 9, pp. 355–364, May 1983.

[61] J. Y. Chung, J. W. S. Liu and K. J. Lin, "Scheduling Periodic jobs that allow imprecise results ," *IEEE Transactions on Computers*, 39, pp. 1156–1173, September 1990.

[62] K. J. Lin, S. Natarajan, J. W. S. Liu and T. Krauskopf, "Concord : A system of imprecise computations," in *IEEE Proceedings on Compsac.* Japan: October 1987.

[63] K. J. Lin, S. Natarajan and J. W. S. Liu, "Imprecise results : Utilizing partial computations in real-time systems," *IEEE Proceedings on Real-Time System Symposium*, 8, pp. 210–217, 1987.

[64] J. W. S. Liu, K-J. Lin, W-K. Shih, A. C-S. Yu, J-Y. Chung and W. Zhao, "Algorithms for Scheduling Imprecise Computations," *Computer*, 24, no. 5, pp. 58–68, May 1991.

[65] C. M. Krishna and K. G. Shin, "On Scheduling Tasks with a Quick Recovery from Failure," *IEEE Transactions on Computers*, 35, no. 5, pp. 448–455, May 1986.

[66] A. Y. Wei, K. Hirachi, R. Cheng and R. H. Campbell, "Application of the Fault-Tolerant Deadline Mechanism to a satellite on-board computer system," *10th International IEEE Conference on Fault-Tolerant Computing*, 10, pp. 107–109, 1980.

[67] A. L. Liestman and R. H. Campbell, "A Fault-Tolerant Scheduling problem," *IEEE Transactions on Software Engineering*, 12, no. 11, pp. 1089–95, Nov 1986.

[68] D. P. Siewiorek, V. Kini, H. Mashburn, S. R. McConnel and M. M. Tsao, "A Case study of C.mmp, C.m* and C.vmp - Part I - Experiences with fault-tolerance in multiprocessor systems," *Proceedings of IEEE*, 66, no. 10, pp. 1178–1199, Oct 1978.

[69] D. P. Siewiorek and R. S. Swarz, *The theory and practice of Reliable System Design*. Digital Press, 1982.

[70] T. Anderson and P. A. Lee, *Fault Tolerance: Principles and Practice*. Englewood Cliffs, NJ, Prentice-Hall, 1980.

[71] O. B. V. Linde, "Computers can now perform vital functions safely," *Railway Gazette International*, 135, no. 11, pp. 1004–1006, November 1979.

[72] S. K. Shrivastava, "Sequential Pascal with Recovery Blocks," *Software-Practice and Experience*, 8, pp. 177–185, 1978.

[73] H. Sullivan and T. R. Bashkow, "A large scale, homogeneous, fully distributed parallel machine I," in *Proceedings 4th Symposium on Computer Architecture*, vol. 4. pp. 105–117, March 1977.

[74] J. Rattner, "Concurrent Processing: A New Direction In Scientific Computing," in *AFIPS Conference Proceedings*, vol. 54. pp. 157–166, 1985.

[75] M. T. Heath, "The hypercube: A tutorial overview ," in *Proceedings 1st Conference on Hypercube Multiprocessors*, vol. 1. Knoxville, TN: pp. 7–10, September 1985.

[76] F. Gray, Pulse Code Communication (US Patent 2 632 058), March 1953.

[77] E. N. Gilbert, "Gray Codes and Paths on the n-cube," *Bell System Journal* , 37 , pp. 815–826, May 1958 .

[78] J. R. Bitner, G.Ehrlich and E. M. Reingold, "Efficient Generation of the Binary Reflected Gray Code and its applications," *Communication of ACM* , 19 , pp. 517–521, 1976 .

[79] E. M. Reingold, J. Nievergelt and N. Deo, *Combinatorial Algorithms Theory and Practice.*. Englewood Cliffs, NJ, Prentice-Hall, 1977.

[80] S. G. Akl, *Design and Analysis of Parallel Algorithms.* Englewood Cliffs, NJ, Prentice-Hall, 1989.

[81] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications.* Englewood Cliffs, NJ, Prentice-Hall, 1988.

[82] D. E. Knuth, *The Art of Computer Programming : Semi Numerical Algorithms*, vol. 2. Reading, MA, Addison Wesley, 1973.

[83] R. Sivakumar, "VLSI Implementation of a Router for the Backtrack-to-the-origin-and-retry-routing scheme of the hypercycle based interconnection networks," Univ. of Victoria, MASc Thesis, 1991.

VITA

Surname:	Srinivasan	Given Names:	Anand
Place of Birth:	Coimbatore, India	Date of Birth:	9th July 1965

Educational Institutions Attended:

University of Victoria	1989 to 1991
Jawaharlal Nehru University, India	1986 to 1989
University of Delhi, India	1983 to 1986

Degrees Awarded:

MCA	Jawaharlal Nehru University	1989
BSc (Hons)	University of Delhi	1986

Awards :

University of Victoria Fellowship 1989 to 1991

Publications:

1. A. Srinivasan and G. C. Shoja, "A Fault-Tolerant Scheduler for Distributed Real-Time Systems" *Proceedings of the 1991 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, Canada, vol. 1, May 1991, pp. 219 - 222 .

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis: Fault-Tolerant Distributed Real-Time Scheduling

Author: ______________________
Signature

Anand Srinivasan
____
(Name in Block Letters)

August 9, 1991
____
(Date)