Fair-Share Scheduler for Computing and Telecommunication Systems

By

Qi Wang

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science
Information and Systems Science

School of Computer Science
Carleton University
Ottawa, Ontario, Canada K1S 5B6

December 2003

Your file Votre référence
ISBN: 0-612-89868-7
Our file Notre référence
ISBN: 0-612-89868-7

Abstract

ChorusOS is a real-time operating system. The first part of the thesis focuses on allocating a fair share of the CPU resource to multiple processes that are running concurrently. Based on the native ChorusOS scheduling policies, we examine four different schedulers that distribute the CPU resource to three classes of jobs: the Equal-Priority scheduler, the Static scheduler, the Dynamic-Priority scheduler, and the Adaptive scheduler. The experimental results have shown that in general the Adaptive scheduler performs better than the other three.

In the second part of the thesis, sharing of both CPU and network resources are investigated. We apply these scheduling schemes to the performance prototype of a network router. In addition to CPU scheduling, three different packet-dropping policies are also examined: the Default-Link, the Fixed-Link, and the Dynamic-Link. The experimental results show that the Adaptive scheduling policy performs reasonably well in terms of fairness, and the Dynamic-Link dropping policy provides more flexibility.

Acknowledgments

I would like to express my sincere thanks to my supervisors, Professor Shikharesh Majumdar and Dr. Chung-Horng Lung, for the valuable guidance, encouragement and effort they put into this thesis over a long period of time.

I would like to thank Dr. R. Gregory Franks for his help and support during my research at Nortel. And I want to thank all the faculty and staff at the School of Computer Science and the Department of Systems and Computer Engineering, as well as the School of Mathematics and Statistics in Carleton University, for everything I have learned here.

I would also like to thank CITO and Nortel Networks for providing the financial support for this research.

Finally, I give my special thanks to my wife Xiuying Jin, my daughter Mingyu, my son Patrick and my parents for their endless love and encouragement.

TABLE OF CONTENTS

List of Tables

List of Figures

Glossary of Terms

ADR	Average Difference Ratio
BSI	Base Scheduling Interval
CBQ	Class-Based Queuing
CBWFQ	Class-Based Weighted Fair Queuing
ChorusOS	Chorus Operating System
COTS	Commercial-off-the-Shelf
CSFQ	Stateless Fair Queuing
DR	Difference Ratio
FCFS	First Come First Served
FIFO	First In First Out
FSS	Fair-Share Scheduling
GCU	Group CPU Utilization
IP	Internet Protocol
ISP	Internet Service Provider
LAN	Local Area Network
MPLS	Multi-protocol Label Switching
OH	Overhead
OSPF	Open Shortest Path First
PQ	Priority Queuing
QoS	Quality of Service

RTOS	Real-Time Operating System
SFQ	Start-time Fair Queuing
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WFQ	Weighted Fair Queuing
VGU	Virtual Group Utilization

Chapter 1 Introduction

As more and more people have been accessing the Internet over the past decade, the volume of network traffic has been increasing dramatically. This increase in volume has driven the telecommunication industry to put a great deal of effort into developing telecommunication equipment to speed up the processing. Internet Service Provider (ISP) and telecommunication product developers have introduced customers to many new products that feature the latest technologies, ranging from high-speed backbones, fast routers and switches to prime end-user equipment. Today's Internet is faster, produces more colorful information, and is more efficient at gaining access for most interested users.

Most pieces of telecommunication equipment have the ability to run multi-tasking applications. Each running task is designed to perform a different job requirement. Usually these tasks will share all the resources of the piece of equipment. To enable these tasks to run efficiently and to perform optimally, a process scheduler is required for each of these pieces of equipment in order to distribute the resource to the correct process at the right time.

These applications typically run on top of various real time operating systems (RTOS). In addition to the vendor-designed operating systems, there are also several Commercial-off-the-Shelf (COTS) RTOSs available for telecommunication companies, including QNX, VxWorks, and ChorusOS.

1.1 Motivation

Process scheduling becomes crucial in a multi-tasking environment. Within a multi-process multi-threaded single server, there are different groups of processes, with each process responding to different requirements. Each process needs a certain amount of resources to finish its job. Most commonly, these processes share the same resources within the system, such as the CPU. Thus, the distribution of these resources among processes has become more important for achieving a higher overall system performance.

ChorusOS was developed by the French research institute INRIA, and has been included in the Sun Embedded Workshop since 1997. It was understood to be an attractive real-time operating system on which to run telecommunication applications. It provides high performance and high availability with a simple, flexible configuration mechanism [10]. This thesis was motivated by Nortel Networks, which was interested in resource management for telecommunication applications such as switches and routers running on top of ChorusOS.

"In the Internet model, constituent networks are connected together by IP datagram forwarders which are called routers or IP routers" [5]. A router is a device that distributes packets over the network. Each router within the network acts as a server node and has a routing table that stores the network topology information. The node gets a packet from an edge switch (could be an edge router, or a direct source) or from the other router, retrieves the destination information from the packet and topology information stored in the routing table, and forwards it to the appropriate router or edge switch. Various types

of packets are transmitted over the network. Among these diversified packets, each packet has a different service requirement: real-time or non-real-time; urgent or normal. The design of a network router must meet the time constraints for the highest number of packets possible. As a case study, this thesis investigates the application of appropriate scheduling strategies for achieving high performance routers.

1.2 Goals

ChorusOS was proposed for commercial use, and the characterization of its performance is important not only for Nortel but also for other telecommunication system developers. In the process of this research, we aim to get a clear understanding of the scheduling policies provided by ChorusOS. In order to provide a fair share of the CPU resource to all the processes, a telecommunication application is investigated. The main goals of the investigation into fair share scheduling are summarized below:

- Design and implement effective schedulers to conduct a common Fair-Share Scheduling (FSS) for a multi-process multi-threaded application running on a uni-processor machine, by using the scheduling policies available from ChorusOS.
- Develop Fair Share Scheduling strategies for a network router and investigate their performance under different workloads.

1.3 Contributions

The contributions of the thesis are summarized briefly as follows:

- The thesis provides a characterization of the different scheduling policies of ChorusOS running on a single processor. It reveals the inter-relationship of concurrent multiple processes with different scheduling policies.
- Four schedulers that distribute the CPU resource to three classes of jobs are developed: Equal-Priority, Static, Dynamic-Priority and Adaptive. The performances of these schedulers are investigated under various workloads. Of these four, the Adaptive scheduler seems to be best suited for satisfying the Fair-Share requirement.
- A number of different scheduling policies and packet-dropping policies are proposed for a network router. A prototype-based investigation is used to analyze the performance of these scheduling policies.

1.4 Thesis Outline

Chapter 2 gives a detailed review of ChorusOS and its scheduling policies. It also reviews the scheduling policies of a network router and other common Fair Share-related schedulers available in the literature. Chapter 3 first characterizes the scheduling policies of ChorusOS. Four schedulers are then developed for performing Fair Share Scheduling for a common multi-tasking system over ChorusOS. Chapter 4 describes the performance prototype of a network router on top of the ChorusOS. It also presents an investigation of several scheduling policies and packet-dropping policies to achieve better router performance. Chapter 5 provides the conclusion and discusses the directions for future research.

Chapter 2 Background and Literature Review

A network router that serves as a typical telecommunication application is introduced in this chapter. The chapter also discusses the ChorusOS operating system and reviews the Fair-Share Scheduling techniques. Following the overview of a network router in the Section 2.1, the ChorusOS system architecture and process scheduling policies are reviewed in Section 2.2. Section 2.3 gives a brief description of the Fair Share Scheduler, and Section 2.4 reviews the various general policies related to Fair Share Scheduling. A summary is presented in the last section.

2.1 Network Router

There are basically two different types of network routers: a software-based router and a hardware-based router. While hardware-based routers were developed to support fast packet forwarding through increased speed, software-based routers are playing an important role in the way that they provide various service functions with flexibility for future expansion [5]. This thesis focuses on the software-based router type.

2.1.1 Overview of Structure

Within a traditional software-based router, the packet-switching software application runs on a general-purpose CPU. Each router has at least two connections over the network. One route database (routing table) resides within each router. The IP packet is forwarded based on the relevant information in the routing table, either to another

router or to the destination host. Figure 1 shows the process structure of a network router [18].

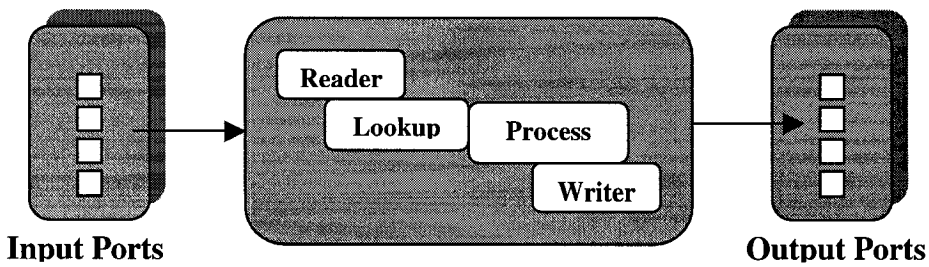


Figure 2-1 Process Structure of Network Router

The application execution within each router may have a single or multiple processes to complete the following functions [18]:

- *Reader*: gets the packet from each input port, and puts it into the queue linked to the lookup function;
- *Lookup*: looks up the routing table information and determines the output port for the packet;
- *Process*: performs the necessary computation based on the packet; and
- *Writer*: sends the packet to the determined output port.

2.1.2 Overview of Queuing

In the network engineering context, queuing refers to the act of storing packets or cells where they are held for subsequent processing [17]. Within a network router, queuing happens when the *Reader* receives packets from the input ports, or may occur before the *Writer* sends packets out to the output ports. Queuing is the crucial component

of a router, where a number of asynchronous processes are bound together to switch packets via queues. Queuing is also critical to quality of service (QoS) to manage various queuing mechanisms based on the QoS level. This section describes five existing different queuing strategies and focuses on output queuing.

- FIFO (First In, First Out) Queuing

  The packets from the input ports are sent to the output ports in the order that they are received. FIFO Queuing is considered to be the standard method for the network router, as long as the router operates at a sufficient level of transmission capacity and an adequate level of switching capacity. However, FIFO can cause significant queuing delays when the network loads increase to a certain level and may result in a packet loss [4].

- Priority Queuing (PQ)

  The packets from the input ports are ordered according to user-defined criteria that classify the placement order for each type of packet. High-priority packets are always put at the front of the output queue, and before the lower priority packets [4].

  It is possible to have multiple levels of priorities to serve different packets, which gives more flexibility in designating the order of preference for each group of packets. On the other hand, this flexibility adds more computational overhead and may have a significant impact on the packet-forwarding performance. It could also lead to starvation for the lower priority group of packets.

- Class-Based Queuing (CBQ)

This is a variation of priority queuing with multiple output queues. CBQ is intended to prevent the starvation of packets in the lower priority group. The service provider can specify the rules for processing various types of packets in order to achieve the desired preference for different groups of packets.

The resource distribution is useful for high-priority packets, but it means that the low-priority packets are transmitted at a lower rate as well. CBQ is considered to be a simple method for providing link sharing for various classes of services [8]. However, the computational overhead limits its usefulness for providing differentiated classes of service with high-speed links.

- Weighted Fair Queuing (WFQ)

  By giving each traffic flow a different weight, the entire bandwidth of the network link is distributed among all the active data flows [4]. WFQ tries to prevent buffer starvation and give a predicable response time. The bandwidth is shared in a proportional manner, and the sharing takes place either at a single level or a hierarchical multi-level [1].

  WFQ sorts the incoming packets into separate flow queues and sends out a fixed portion for each flow at a time. The bandwidth is distributed into equal shares for each active flow, and the lowest volume flow finishes the process first. In this way, WFQ prevents longer flows from consuming network resources that could starve shorter flows. Similar to CBQ, the computation overhead of WFQ is the major obstacle to its scalability.

- Class-Based Weighted Fair Queuing (CBWFQ)

  CBWFQ is a combination of CBQ and WFQ. CBWFQ assigns a fixed amount of bandwidth to a class of packets [3, 4]. The queuing within the same class of packets employs WFQ sorting; that is, it uses flow-based queuing. CBWFQ is the latest queuing technology used on a network router for QoS.

2.1.3 Overview of Scheduling

With different queuing strategies applied in the network router, the Round Robin scheduling policy is usually selected to serve each packet queue. That is, the *Writer* gets a fixed portion of packets from each queue, and then sends them out to the destination.

Goyal [12] proposes a Start-time Fair Queuing (SFQ) algorithm that is used to serve the packets in Integrated-Services-Packet-Switching-Networks. Each packet is stamped with a computed start tag upon its arrival. Based on the start tag, a computed finish tag is also attached to the arrived packet. This finish tag will be used to compute the start tag of the next packet within the same flow. The scheduler serves the packets in the increasing order of their start tags. The algorithm provides fairness for video and data applications, as well as a reasonable delay for low throughput applications.

As a resource allocation algorithm, SFQ can be also used for fair CPU resource allocation [11]. By assigning a start tag to each thread and scheduling the threads in an increasing order of the start tags, the system can execute these threads towards the fair consumption of CPU resources among multiple applications.

Cobb [6] explores a Time-Shift scheduler that is used to forward packets from multiple input flows to a single output channel. The scheduler has one separate queue for

the incoming packets from each flow. When each flow is queued fairly, the first packet in each queue is time-stamped based on its proportional-delay rate, and the scheduler always starts processing the packet with the shortest delay.

Proportional Share scheduling is used to achieve QoS in the network router via the following characteristics: each process runs at a fixed cycle rate; that is, it has a fixed portion of the resources. If one process is in the idle state, its unused capacity is distributed among all the active processes, and its unused share is lost without any compensation [7].

Bennett [1] proposes a hierarchical packet fair-queuing algorithm called WF^2Q+. Using a general WFQ algorithm, the network router selects the next packet for service at scheduling time. The packet with the smallest finish time tag in all the packet queues will be selected. With WF^2Q+, the network router performs the selection one scheduling time ahead. This means that the router will transmit the previously selected packet and pick the packet with the smallest virtual finish time to be the next packet transmitted. It provides the smallest delay bound and fair bandwidth distribution.

Bodamer has proposed a Weighted Earliest Due Date scheduling algorithm, which provides differing delay for real-time traffic in a network with service differentiation [2]. Each packet is set a service deadline as it arrives. The packets are served in increasing order of their service deadlines. If two classes of packets run into competition, the router will serve the packet with the higher weight in the class first.

Shreedhar explores a variation of round robin scheduling that serves network packets [20]. This proposed scheduler is different from the general round robin scheduler

in the way it processes large-sized packets. If a packet was not sent out in the previous service round due to its larger size, compensation is given to the queue to which the packet belongs. The experimental results show its feasibility for implementing Fair-Queuing in network routers.

All of the schedulers described above maintain a per-flow state at each network router. Stoica proposes a Stateless Fair Queuing (CSFQ) used for core routers. By adding per-flow labeled information into the header of each packet, the CSFQ scheduling achieves bandwidth allocation which is approximately fair [22].

2.2 Overview of ChorusOS

ChorusOS provides a complete host-target development environment for its users. This enables the user to develop an application on a host system and execute it on a reference target board where the ChorusOS is running [23, 25].

2.2.1 System Architecture

ChorusOS uses a layered, component-based architecture that gives the user maximum flexibility. The runtime instance of the operating system can be configured specifically for different services in order to meet the individual needs of a given user application or system environment. Figure 2-2 shows the component-based architecture of ChorusOS.

Each run time environment can be built with a combination of any of the following components [23]:

- Core Executive (Micro-kernel)

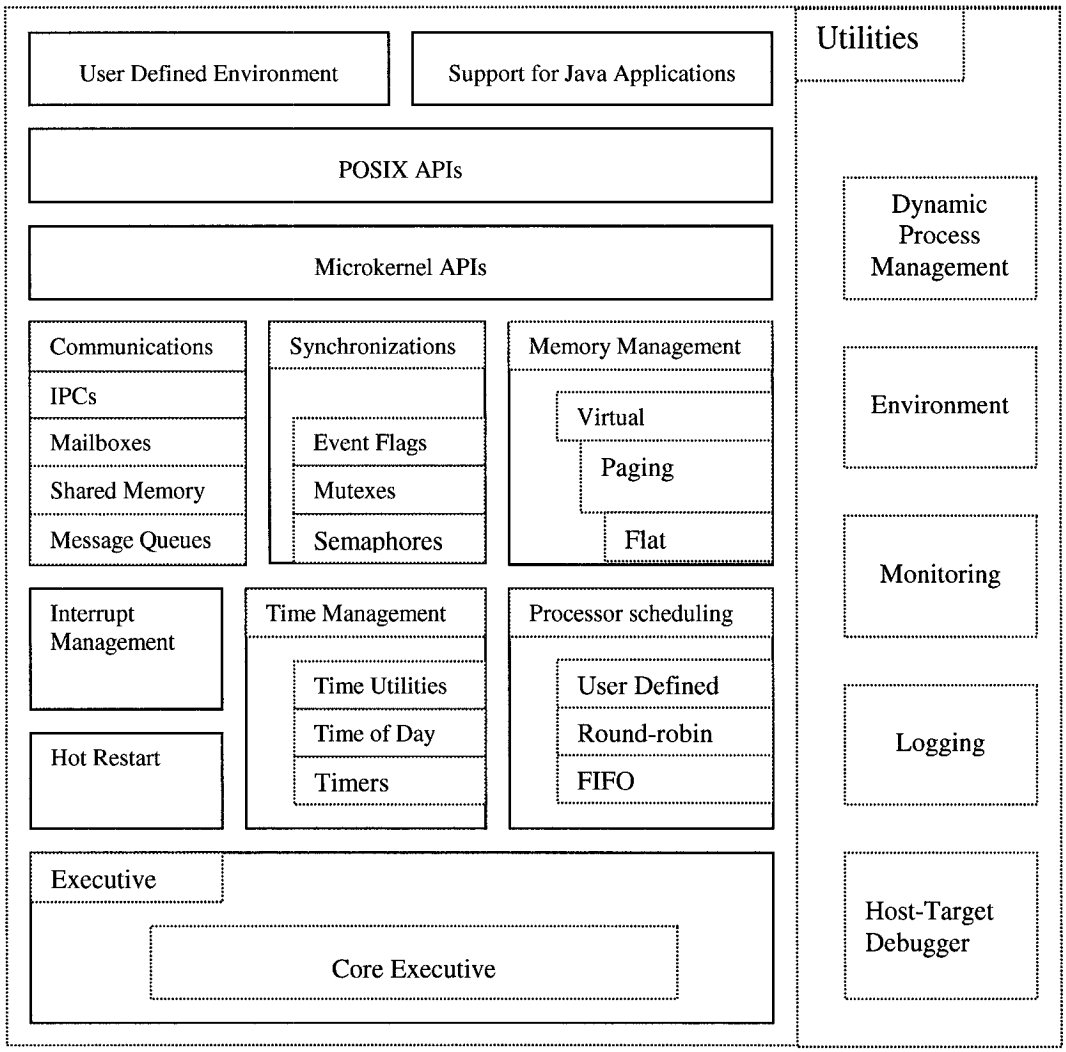


Figure 2-2 Component-based Operating System Architecture (from [23])

This provides the fundamental functionality, which supports:

— Multiple independent real-time applications

— User and system applications in different address spaces

- Memory Management

  This provides three different management models for serving various memory allocation requirements.

- Hot Restart and Persistent Memory Management

  This supports rapid reloading and re-initializing. When the system detects crashes in re-startable actors, it automatically restarts them from an actor image retained in persistent memory.

- Synchronizations

  This component provides different shared structures and access controls for threads running within the same application.

- Time Management

  This provides multi-level timing services that range from interrupt-level one-shot time-out services, to high-level interval timing, universal timing, to a real-time clock.

- Communications

  This component allows processes to communicate and synchronize with each other through the passing of messages, whether they are in the same site or not.

- Processor Scheduling

  Unlike the UNIX time-sharing system, which provides resource-sharing among multi-level priority processes [27], ChorusOS provides simple pre-emptive scheduling based on thread priorities.

- APIs

  These provide microkernel APIs and POSIX APIs, which are available for applications to interact with the system effectively.

- Utilities

This component provides several utilities that can be added for managing the system and applications on the target.

2.2.2 Scheduling Policies

CPU scheduling on ChorusOS uses priorities on a per-thread basis. The core scheduler within the ChorusOS micro-kernel performs pure pre-emptive scheduling. The micro-kernel keeps track of the priority of each active thread and runs the one with the highest absolute priority [23, 25].

On top of the micro-kernel, ChorusOS provides different scheduling classes. Each class communicates with the core scheduler and makes its own scheduling decisions based on its own class attributes and behaviors. With all the scheduling policies, ChorusOS maintains one queue for every single priority value.

With version 4.0, ChorusOS provides two mutually exclusive priority-based schedulers: SCHED_FIFO and SCHED_CLASS. The micro-kernel implements the SCHED_FIFO, while SCHED_CLASS gives the user the flexibility to manipulate the processes. The characteristics of each scheduler are summarized as follows:

SCHED_FIFO scheduler

This is the default scheduler for the system. It has the following characteristics:

- The thread priority varies from the highest value of 0 to the lowest value of 255.
- A ready-to-run thread is always inserted at the end of its priority queue.
- A running thread is pre-empted only if there is a higher priority thread ready to run.

- The pre-empted thread is placed at the head of its priority queue.

SCHED_CLASS scheduler

The SCHED_CLASS scheduler implements four scheduling policies:

- CLASS_FIFO

  CLASS_FIFO behaves in the same way as the SCHED_FIFO scheduler.

- CLASS_RR

  CLASS_RR has a priority-based pre-emptive scheduling with round-robin time slicing. The thread priority varies from a high of 0 to a low of 255. A ready-to-run thread is always inserted at the end of its priority queue. Threads in a given priority queue are run in a round-robin fashion. The selected running thread is given a fixed time quantum. If a thread is still running when its time quantum expires, it is blocked and will be placed at the end of its priority ready queue. The time quantum is the same for all the threads with different priority levels. A running thread is pre-empted only if there is a higher priority thread that is ready to run, and the pre-empted thread is placed at the head of its priority queue.

- CLASS_RT

  CLASS_RT has the same policy as the real-time class of UNIX SVR4.0, which is a CLASS_RR policy with per-thread time quantum.

  The thread priority values of CLASS_RT are in a narrower range than those of CLASS_FIFO and CLASS_RR. That is, the value varies from a high of 159, which corresponds to 96 of CLASS_FIFO and CLASS_RR, to a low of 100,

which corresponds to 155 of CLASS_FIFO and CLASS_RR. Each thread may have a different scheduling time quantum that varies from seconds to nanoseconds [24].

In order to support POSIX, ChorusOS also provides three kinds of POSIX schedulers: SCHED_FIFO, SCHED_RR and SCHED_OTHER. SCHED_OTHER is the same as SCHED_RR.

- The SCHED_FIFO and SCHED_RR policies are the same as the policies of CLASS_FIFO and CLASS_RR respectively, except the priority schemes are reversed. In other words, with CLASS_FIFO and CLASS_RR, the higher the priority value, the lower the priority; for SCHED_FIFO and SCHED_RR, the higher the value, the higher the priority.

2.3 Fair-Share Scheduler

Traditionally, fair-share schedulers allow the resources to be shared fairly among processes. A scheduler must achieve the following objectives in order to be considered fair:

- Each user should eventually get its entitled predetermined share of the CPU.
- A user consuming a relatively higher share of the CPU resource compared to its predetermined share should receive a lower opportunity to occupy the CPU, while the user which consumes a relatively lower share will have more chance at getting the CPU resource.

- If one user is not going to use its share of the CPU resource within a specific time period, all the other active users can share this additional CPU resource. This means a user can get more CPU share if there is no other user using the machine.

2.4 Related Work on Fair-Share Schedulers

In a non-pre-emptive time-sharing system like UNIX, process scheduling provides a way of adjusting the priority of each job so that its running frequency and execution time quantum will be changed accordingly [21]. The system has an aging mechanism that can process all the jobs fairly over a long runtime. Adding an additional job will affect all the other active jobs within the system, because the CPU resource must be re-distributed among all the jobs [15]. This section will review some of the existing fair-share schedulers.

2.4.1 Proportional Share

Proportional share schedulers assign the CPU resource to each process so that each process receives a CPU share that is less than or equal to its prescribed share [13, 19]. Knowing the rate requirements of each process, a proportional scheduler can distribute the resources accurately in order to meet the deadline of each individual process.

Epema [7] examines several Proportional-Share Scheduling policies for a multi-processor single server and a multi-server environment. Four different scheduling policies are described:

- Priority Queuing

Jobs in a group have higher priority than jobs in another group if the first group requests a greater CPU share than the second group.

- Processor Sharing

  All jobs in all groups receive service at the same rate.

- Priority Processor Sharing

  Each job has a service rate proportional to the CPU share required by its group.

- Group Priority Processor Sharing

  The group service rate is proportional to its required CPU share.

With Priority Processor Sharing and a uni-processor single server, each group of jobs has a required CPU share. If all the groups of jobs require resources within a specific period of time, each group of jobs will receive a proportional amount of share closest to their required share. If the jobs in one group do not require their entitled predetermined resources, these resources will be shared by other active groups. These resources are allocated to each active group proportional to their required share.

2.4.2 Hierarchical Share

Hierarchical scheduling techniques are used to adapt multi-level scheduling to soft real-time systems [9, 11, 14]. By splitting the scheduling decisions into multiple levels, this architectural approach creates flexibility for meeting different needs when running a mixture of applications. However, choosing the ideal configuration is not a trivial process.

Kay [14] introduces "A Fair Share Scheduler" that distributes resources fairly among several groups, according to their individual requirements. The scheduler initially addressed the problem of CPU allocation within a student environment, and allocated the resources of a single machine to multiple users from multiple organizations.

Sharing the same machine, each group was given its own predefined share of the CPU to which it was entitled in order to run its program, and each user within a group had its own share as well. Each user could have more than one process running in the system. For each user there is a history of the amount of CPU resource that it actually used, which is called decayed usage. The decayed usage of each user is related to the priority of each process the user runs on the machine. The lower the priority of the process running, the less the decayed usage attributed to its user. Decayed usage also relates to the running time of each process. If a user runs a process during the peak busy period, its decayed usage is higher. When the same process is run in a time other than the peak period, the decayed usage is lower.

Two levels of scheduling policies are introduced in this scheduler: user-level and process-level. At the user-level, the scheduler accumulates the decayed usage of all the active users and provides an estimated usage for each active user in order to indicate its expected CPU usage in the next scheduling time period. At the process level, each time the running process releases the CPU, the scheduler first obtains the CPU usage for the user of the process. The scheduler then adjusts its priority based on the user's CPU usage, CPU share, and the total number of active processes for that user. In other words, if a user runs more than one process at the same time, and the user has already used its predetermined CPU share, then each process that belongs to this user will be scheduled at

a lower priority. If a user runs fewer processes but has not used its predetermined share, the processes belonging to this user will be scheduled at a higher priority. At the end of each scheduling period, the scheduler will increase the priority of all the ready processes in order to increase their opportunities to run.

This scheduler achieves true fairness among users and between groups. It provides users with useful information so that they can get an estimate of the response time of the process they are planning to run, as well as the cost of running it. This helps users spread their workloads, and achieve the best performance possible [14].

However, this scheduling policy only applies to a UNIX-alike time-sharing system, where all processes with different priorities can share the CPU resource in a proportional way. This policy does not apply to the ChorusOS, as ChorusOS implies pre-emptive priority scheduling. The thread will always run if there is no other thread with a higher or same priority in the active state.

2.4.3 Lottery Scheduler

Waldspurger [26] proposes a "lottery scheduler" to be used for the fair allocation of resources. It is basically a proportional-share resource management scheduler, but it operates more efficiently. Lottery scheduling is a randomized resource allocation mechanism. There are two important notations in Lottery scheduling:

- *Tickets:* stands for the resource rights held by an active client (or process).
- *Lottery:* stands for the selection of an active process ready to consume the resource.

Each active client has its own share of the resource. Every time the scheduler runs, it assigns each active client a number of tickets corresponding to its share and calculates the total number of tickets for all active clients. It then generates a random number that falls between 1 and the total number of tickets. The client whose ticket matches this random number is chosen to win the lottery and is allocated the resource.

It is expected that lottery scheduling will achieve fairness in the long run. However, it may not fit the needs of QoS, as some of the applications may not get guaranteed service at a specific point of time.

2.5 Summary

This chapter presented a literature review on the related areas. The review consists of the following three different components.

- Network Router

  The chapter reviewed the structure of a network router, its various queuing policies and the scheduling policies. Five different queuing policies were presented: FIFO, PQ, CBO, WFQ, and CBWFQ. For the scheduling policy, round-robin is widely used.

- ChorusOS

  The chapter briefly reviewed the component structure of ChorusOS. The chapter also conducted a detailed review of the scheduling policies of ChorusOS, which include native scheduling policies and POSIX scheduling policies.

- General Fair Share Scheduler

The chapter reviewed the various fair-share-scheduling policies in non-pre-emptive systems, ranging from proportional scheduling to hierarchical scheduling.

This chapter also stated the characteristics of a fair share scheduler used in this thesis.

Chapter 3 Fair-Share Scheduler

This chapter describes a fair share scheduler which runs on the ChorusOS. Section 3.1 characterizes the scheduling behavior of the ChorusOS. The workload used in the investigation of the schedulers is described in the following section. Section 3.3 introduces the metrics used to measure performance. Section 3.4 examines four different scheduling policies. The results of the experiments are described in Section 3.5. The last section provides a summary of the implementation of the fair share scheduler.

3.1 Characterization of the Scheduling Behavior of the ChorusOS

Five different scheduling policies are provided within the ChorusOS: CLASS_FIFO, CLASS_RR, CLASS_RT, POSIX SCHED_FIFO, POSIX SCHED_RR. However, the inter-relationship of these scheduling policies is not clearly documented. Motivated by Nortel Networks that was interested in obtaining a clear vision of ChorusOS scheduling policies, an experimental multi-task application has been developed.

This application has been designed with a group of threads that use mixed scheduling policies running concurrently on a single machine. A 133MHz Pentium PC is used as the running target. The scheduling behavior of ChorusOS is not expected to depend on the hardware platform used.

3.1.1 Test Application

The application is implemented in order to find the execution sequence of a number of threads that use different scheduling policies when they are run concurrently on a single machine. The application starts with the generation of a number of threads, and then sets the scheduling policy and priority value for each of these threads. The main thread then initializes a semaphore, and waits for the signaling of that semaphore. All the threads are created to function in the same way: they consume a certain amount of CPU, and then signal the semaphore that the main thread is waiting. The pseudo code is shown in Figure 3-1.

```
Main()
{
        Spawn another application as needed for two processes
        Initialize the scheduled parameters
        For 1 to Number-of-threads-to-create
                Create thread
                Setup scheduling policy for the created thread
        End For
        Initialize the semaphore A
        Wait for the signal of the semaphore A
        Exit
}
Thread()
{
        Consume CPU by running a predefined for loop
        Signal semaphore A
        Exit
}
```

Figure 3-1 Pseudo Code of Testing Application

3.1.2 Test Results

Two test cases complete this investigation. The first test was designed to find the execution sequence within one process, and the aim of the second is to find the execution sequence for two processes running concurrently.

In the first case, ten threads are generated within this testing application. Out of these ten threads, eight threads run at the same priority which will be explained in the next section, two for each of the scheduling policies (CLASS_RR, CLASS_FIFO, SCHED_RR, SCHED_FIFO). Of the other two threads running on the CLASS_RT, one has a higher priority than those eight threads, while the other has a lower priority. We set the running time quantum (Time Slice) at 10 and 100 milli-seconds for CLASS_RT and RRs respectively. The execution time for each thread is 23 seconds. The results of the experiment are summarized in Table 3-1.

Table 3-1 Mixed-Thread Execution Sequence within Single Application

Thread #	Class	Priority	TimeSlice(ms)	StartTime(s)	FinshTime(s)
1	CLASS_RT	115	10	208	232
2	CLASS_RT	135	10	0	23
3	CLASS_RR	130	100	23	208
4	CLASS_RR	130	100	23	208
5	CLASS_FIFO	130		23	47
6	CLASS_FIFO	130		47	70
7	SCHED_RR	125	100	70	208
8	SCHED_RR	125	100	70	208
9	SCHED_FIFO	125		70	93
10	SCHED_FIFO	125		93	116

Each column in this table is described in detail. The "Thread #" column shows the creation sequence of the threads. Thread # 1 is created first. All of these threads are

ready to run once the main thread has begun to wait on the semaphore. The "Class" column shows the scheduling policy used by each thread. The "Priority" column shows the priority of the thread within its scheduling policy. The "TimeSlice(ms)" shows the running time quantum used in the Real-Time and round-robin scheduling policies. The "StartTime(s)" column shows the actual start time of each thread from its creation. The thread starts its execution at this time. The "FinishTime(s)" column shows the time at which each thread finished its execution.

In the second case, two identical applications run concurrently as two separate processes. Each process creates six threads, one for each scheduling policy, except that two are for CLASS_RT. Of these two CLASS_RT threads, one has a higher priority, another a lower priority. We set the time slice at the same value as that used in the first test case. The execution time for each thread is 24 seconds. The test results are summarized in Table 3-2.

3.1.3 Analysis of the Results

From the experimental results shown in the last section, we have verified the following ChorusOS scheduling behavior:

- There is a priority value equivalent among all the scheduling policies. That is, one priority value within one scheduling policy is equivalent to the other value within the other scheduling policy [23, 24]. For instance, a priority value 130 for CLASS_FIFO and CLASS_RR is equivalent to the value 125 (255 – 130) for SCHED_FIFO, SCHED_RR and CLASS_RT. As the results in Table 3-1 show,

threads #3, #4 and threads #7, #8 have equivalent priorities. They were run in a Round Robin manner and finished at the same time.

Table 3-2 Execution Sequence for Multiple Applications

Thread #	Class	Priority	TimeSlice(ms)	StartTime(s)	FinshTime(s)
1-1	CLASS_RT	108	10	240	288
1-2	CLASS_RT	113	10	0	47
1-3	CLASS_RR	145	100	47	239
1-4	CLASS_FIFO	145		47	71
1-5	SCHED_RR	110	100	71	239
1-6	SCHED_FIFO	110		71	95
2-1	CLASS_RT	108	10	241	288
2-2	CLASS_RT	113	10	0	47
2-3	CLASS_RR	145	100	95	240
2-4	CLASS_FIFO	145		95	119
2-5	SCHED_RR	110	100	119	240
2-6	SCHED_FIFO	110		120	144

- All the scheduling policies share the same global queues. In fact, in the first test case discussed in the last section, the last eight threads (from #3 to #10) have the same level of priority in the system. Therefore, they were put into the same queue to compete for the CPU resource. Figure 3-2 illustrates the priority queues in which each thread resides. Note that the priority of thread #2 and thread #1 are equivalent to a priority of 120 (255 – 135) and 140 (255 – 115) in CLASS_FIFO, respectively.

From the results shown in Section 3.1.2, we can also conclude the following concerning the scheduling behavior of the ChorusOS:

Priority Queue (120)	#2 (RT-135)
Priority Queue (130)	#3,#4,#5,#6,#7,#8,#9,#10
Priority Queue (140)	#1 (RT-115)

Figure 3-2 Initial Thread Execution Queue

- There are at most 256 process queues in the ChorusOS. Thread #2 in both applications has the highest priority and executes as soon as it is ready to consume its share of the CPU resource. Thread #1 has the lowest priority and can consume CPU resource only when the other threads have finished executing.
- Each thread runs according its own scheduling policy. For example, as shown in Figure 3-2, threads #5, #6, #9 and #10 all operate with the FIFO scheduling policy, and once they get a chance to execute, they run until they are finished. Using the round-robin scheduling method, threads #3, #4, #7 and #8 each run a fixed time quantum and relinquish the CPU, and are then inserted at the rear of the queue. Although threads #3, #4, #7 and #8 start at different times, they finish at about the same time.
- Table 3-2 shows that threads with the same scheduling policy behave the same, even though they belong to different processes. For example, thread 1-6 and thread 2-6 running under SCHED_FIFO with a priority of 110 run in sequence. Thread 1-6 complete completes 24 seconds after it starts execution, where as thread 2-6 starts after thread 2-5 relinquishes the CPU and finishes after 24 seconds. Note that thread 2-5 is scheduled using SCHED_RR receives only a single quantum of CPU time when it starts running at time 119. Similarly thread

1-3 and thread 2-3 receive a single quantum of service before they relinquish the CPU to thread 1-4 and thread 2-4 respectively. Note that all these threads have the same equivalent priority.

3.2 Workload

The workload used in the investigation of the schedulers is described in this section. The schedulers for these applications are described in Section 3.4.

3.2.1 Assumptions

The following assumptions have been made throughout the development of the Fair Share Scheduler:

- There are three groups (1, 2, 3) of jobs running within the system, and each group has its own pre-defined CPU share.
- In order to reflect the differences among the groups, there is a different number of jobs in each group: five jobs in group 1, four in group 2 and three in group 3. All of these jobs run cyclically and each cycle is characterized by the following two phases:

  Computing Phase: Consume CPU time that models the execution of some task

  Thinking Phase: Sleep for a given interval of time to let other jobs run.

  The computing phase is followed by the thinking phase in a cycle. The pseudo code for the scheduled job is shown in Figure 3-3.
- When the CPU is fully utilized, the CPU time consumed by all three job groups should correspond exactly to their allocated share.

```
Begin
        Do Loop
                Burn a given amount of CPU time
                Sleep for a fixed time period
        Until being deleted by main thread
End
```

Figure 3-3 Pseudo Code for a Scheduled Job

- When the CPU utilization in one group drops, there are not enough jobs in that group ready to consume the CPU; the other groups which have more jobs to complete should have more chance to consume CPU resource and exceed their share, if needed.

3.2.2 Description of Parameters

This section defines the parameters used for the implementation of the Fair Share Scheduler.

- Share[i] [i = 1, 2, 3]

  The predefined CPU share for group i. Throughout this study, the shares are distributed as: Share[1] = 50 %, Share[2] = 30%, and Share[3] = 20%.

- Exec[i] [i = 1, 2, 3]

  The execution time in ms for the jobs within group i.

- Slp[i] [i = 1, 2, 3]

  The think time or sleep time in ms for the jobs within group i. A job alternates between an execution and sleep phase.

- VGU[i] [i = 1, 2, 3]

Virtual group utilization (VGU) for group i when Exec[i] and Slp[i] of group i are given. A group's VGU is the maximum utilization achieved by the processes in the group if the number of processors is unbounded. Assuming the number of jobs within group i is n[i], the VGU of group i is defined as:

$$VGU[i] = Exec[i] / (Exec[i] + Slp[i]) * n[i] * 100\%$$

For example, if the five jobs within group 1 have a computation time of 4 ms and a think time of 12 ms, the VGU of group 1 is:

$$VGU[1] = Exec[1] / (Exec[1] + Slp[1]) * n[1] * 100\%$$

$$= 4 / (4 + 12) * 5 * 100\ \% = 125\ \%$$

As shown in the previous example, the VGU of a group can be more than 100% and not realizable on a single CPU system.

- BSI

  BSI refers to the Base Scheduling Interval for each scheduler. The static scheduler uses this time interval to determine when the scheduler will do the next scheduling, while the Dynamic-Priority and Adaptive schedulers use it to calculate the time interval for their next scheduling times.

3.3 Performance Metrics

This section describes the performance metrics used in the fair share scheduler experiments.

- Group CPU Utilization (GCU[i]) [i = 1, 2, 3]

A group's GCU is the actual CPU share consumed by the group during a specific period. For example, if group 1 consumes CPU for 40 seconds out of the last 100 seconds, the GCU of group 1 is 40 %.

- Average Difference Ratio (ADR)

  With a fixed number of jobs in each group, we have a predefined CPU share for each. Due to the variation of the workload and scheduling policies, the GCU of group i can be different from Share[i]. Let

  $$\Delta S[i] = | GCU[i] - Share[i] |$$

  Difference Ratio (DR[i]) is the ratio of the difference ($\Delta S[i]$) to predefined CPU share:

  $$DR[i] = \Delta S[i] / Share[i] * 100\%$$

  ADR is the average value of DR[i] for the three groups.

  $$ADR = (DR[1] + DR[2] + DR[3]) / 3$$

  Table 3-3 illustrates the calculation of the ADR, achieved with a scheduler.

Table 3-3 ADR Calculation

Group#	Share (%)	GCU (%)	ΔS (%)	DR (%)	ADR (%)
1	50	58	8	16	42
2	30	12	18	60	
3	20	30	10	50	

ADR is used to measure the fairness of each of the scheduling policies. A higher ADR means that the scheduler is less fair, as it means some groups have obtained less (or more) of the share than they were entitled to have. A smaller ADR

indicates a scheduler is fairer, because all the groups have obtained a share close to what they were allocated to use.

- Overhead (OH)

  Every scheduler needs a certain amount of time to perform the job scheduling. This in turn slows down the job execution within the system. The OH examines how the scheduler will affect the overall job execution of the system.

  Assume the total system run time is T0 time units; the CPU has no idle time at all; and all the groups of jobs consume the CPU for T1 units of time. The scheduler overhead is given by:

  $$OH = (T0 - T1) / T1 * 100\%$$

- Throughput

  The throughput for a group is defined as the number of job cycles the group finishes per second. Therefore, if each of the five jobs in group1 runs 10 times per second, the throughput of group1 is 50 jobs/s. The purpose of examining the throughput is to understand the load distribution over the different groups.

3.4 Scheduler Implementation

Four schedulers are presented in this section. The Equal-Priority scheduler is implemented mainly for comparison purposes; the Static scheduler runs periodically; the Dynamic-Priority scheduler runs whenever it needs to run; and the Adaptive scheduler runs after a certain amount of time, which is determined at the previous scheduling time.

The ChorusOS implements prioritized pure pre-emptive scheduling as its core scheduling policy. Therefore, all the schedulers are implemented using the CLASS_FIFO

scheduling policy to achieve Fair Share. That is, a smaller priority value means higher priority, and a greater priority value means lower priority. For all four schedulers, the scheduler (main thread) is always assigned the highest priority to ensure it carries out the scheduling at the designated time.

3.4.1 Equal-Priority Scheduler

It is assumed that all the scheduled jobs have the same priority, so they have an equal opportunity to consume CPU time. When the main thread has finished creating jobs with equal priority, it goes to sleep.

Because all the jobs have the same priority, they are put into a single processing queue. All the scheduled jobs run on a First Come First Served (FCFS) basis. The job at the front of the ready queue is always picked up to run until it completes its computation. After sleeping for a specific amount of time, it becomes ready and is inserted at the end of the ready queue.

As jobs in each group run exactly once every cycle, all the jobs run the same number of times during the scheduling time period. For this reason, the GCU of each group is proportional to its VGU, and does not have any relationship with its predefined group share. For example, if all the jobs are designed to execute for 10 ms after sleeping for 40 ms, as they are running on a single machine, each cycle will last 120 ((5 + 4 + 3) * 10) ms. Therefore, the GCU of each group is given by:

GCU[1] = (5 * 10) / 120 * 100%

= 41.67%;

GCU[2] = (4 * 10) / 120 * 100%

= 33.33%;

GCU[3] = (3 * 10) / 120 * 100%

= 25%.

These are proportional to the VGUs of each group.

Figure 3-4 shows the pseudo code of the main thread corresponding to the Equal-Priority scheduler.

```
Begin
        Initialization
        Create jobs for each group with same priority
        Setup experiment timer for the scheduler
        {Begin scheduler}
        Wait for the timer
        {End scheduler}
        Compute GCU for each single group
        Printout the result
End
```

Figure 3-4 Pseudo Code of the Main Thread for the Equal-Priority Scheduler

3.4.2 Static Scheduler

A static scheduler runs periodically. As the scheduler wakes up, it completes the following steps: computes the GCU for each group first, and lowers the priority of the jobs in the group which is running to normal. It then compares the GCU of each group with its own share, and selects the group of jobs with the smallest ratio of GCU[i] and Share[i] to be the next group to run. Finally it raises the priority of the selected group of jobs to a higher priority value, in order to give them more opportunity to consume CPU. Figure 3-5 shows the pseudo code of the main thread for the Static scheduler.

During the scheduler's sleeping period, all the jobs within the system remain split into two process queues. All the jobs in the selected running group reside in the higher priority process queue, while all the other jobs are put into the lower priority queue. Within each priority queue, all the jobs run on an FCFS basis. If the selected group's jobs are ready to run, they will always get a chance to consume CPU resource.

```
Begin
        Initialization
        Create jobs for each group
        {Begin scheduler}
        Do scheduling loop
                Update the GCU for each group
                Lower priority value of the jobs in the running group
                Do CPU share comparison among the three groups
                Select the group with the smallest ratio of GCU and Share
                Increase priority value for the jobs of the selected group
                Sleep for a time period of BSI to let the jobs run
        Until the experiment finish
        {End scheduler}
        Compute GCU for each single group
        Printout the result
End
```

Figure 3-5 Pseudo Code of the Main Thread for the Static Scheduler

However, if no job in the selected group is ready to run, the jobs within the lower priority queue will get a chance to consume CPU resource. As soon as the job in the selected group is ready to run, however, it will pre-empt the lower priority job which is currently running.

The priority setting for threads in the Static scheduler is shown in Figure 3-6. The main thread has the highest priority, the selected group's jobs have medium priority, and the other groups of jobs are assigned the lowest priority. As mentioned in the previous sections, a lower value means a higher priority.

If the VGU of every group is greater than its group Share, we call the workload a balanced workload. With a balanced workload, the GCU of each group is expected to be close to its Share.

Main thread (scheduler)	140
Selected group of jobs	155
Other group of jobs	160

Figure 3-6 Priorities of Static Scheduler

If the VGU of at least one group is lower than its Share, the CPU resource may not be properly distributed over the groups. For example, given that each of the five jobs of group 1 executes for 1 ms after sleeping for 30 ms, while four jobs in group 2 and three jobs in group 3 each execute for 10 ms after sleeping for 30 ms, the VGU for each group is:

$$VGU[1] = (5 * 1) / (30 + 1) * 100\%$$
$$= 16\%,$$
$$VGU[2] = (4 * 10) / (30 + 10)$$
$$= 100\%,$$
$$VGU[3] = (3 * 10) / (30 + 10) * 100\%$$
$$= 75\%.$$

The scheduler will always give a higher priority to the jobs in group 1 as its GCU is much less than its Share. However, the jobs in group 1 may not be ready to consume their share of the CPU resource most of the time, and the jobs in the other two groups will get the opportunity to consume CPU resource even if they have been given the lowest priority. Since the jobs in both group 2 and group 3 are assigned the same priority, they run on an FCFS basis. For both group 2 and group 3, the Static scheduler is similar to the Equal-Priority scheduler, and their group GCUs are proportional to their VGUs. The Static scheduler cannot distribute the CPU resource fairly in this case.

3.4.3 Dynamic-Priority Scheduler

The Dynamic-Priority scheduler is different from the Static scheduler in that when no job in the scheduled running group is ready to consume its share of the CPU resource, the scheduler itself will immediately undertake a rescheduling operation. The idea of the Dynamic-Priority scheduler is to always give the CPU resource to the group with the smallest ratio of GCU to Share, provided that there is a ready-to-run job in the group.

An "invoking" thread is added for the Dynamic-Priority scheduler. The priority of the invoking thread is set between the priorities of the selected group and the other groups, so that the invoking thread will wake up the scheduler when there is no job in the selected group that is ready to use the CPU. The pseudo code of the invoking thread is:

Do loop

    Signal the semaphore that the scheduler is waiting for

Until being deleted by the main thread

A semaphore is implemented for the synchronization of the scheduler and the invoking thread. The priority setting and the relationship between the scheduler and the invoking thread are described in Figure 3-7. The scheduler waits on the semaphore after it has finished scheduling. Once the scheduler is blocked, jobs in the selected group get the opportunity to consume their share of the CPU resource. If no job in the selected group is ready to run, the invoking thread runs and signals the semaphore. Thus the scheduler starts scheduling again.

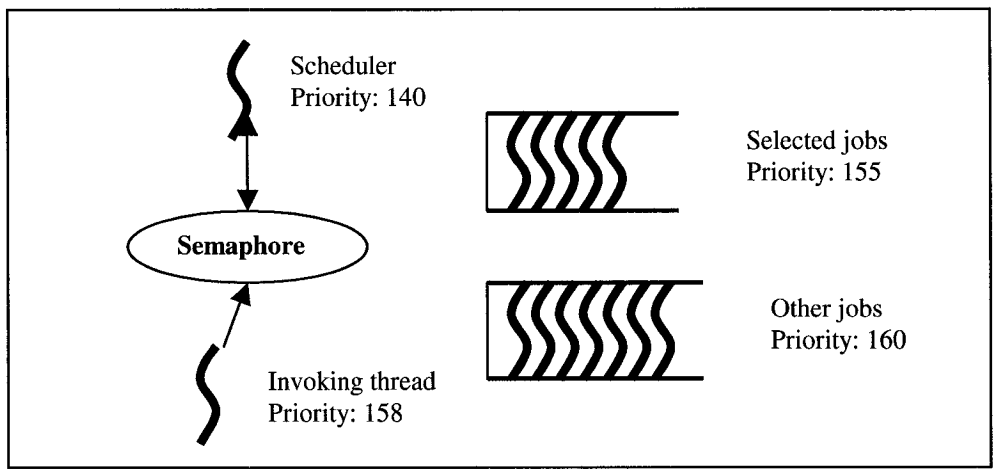


Figure 3-7 Dynamic-Priority Scheduler Structure

The principle underlying the Dynamic-Priority scheduler is explained with the help of Figure 3-7.

- Within a specific scheduling time period, only one group of jobs is selected to run at the highest priority if each group's VGU is greater than or equal to its Share. If there are always some jobs in the selected group which are ready to consume CPU, each group will get the share it is entitled to. The priority value of the jobs in the selected group is raised to the second highest value (155). However, if the

VGUs of more than one group are much lower than their Shares, the scheduler may not distribute the CPU resource fairly. A detailed explanation will be presented later in this section.

- The scheduler will start running if any of the following occurs:

  The jobs within the selected group are busy enough to keep consuming the CPU resource until the scheduled time period is over

  OR

  None of the jobs within the selected group are busy at that moment; then the invoking thread will have a chance to run. It signals the semaphore, for which the scheduler is waiting. If no job is ready to consume the CPU resource during the scheduled period, we call this scheduling overhead an inefficient scheduling time.

- Each time a scheduler starts running, it will compute the GCU for each group. Having compared the GCU of each group to its Share, the group with the smallest ratio of GCU to Share will be selected as the next running group.

Further considerations are needed because when the GCU of at least one group is much lower than its group Share, the scheduler may increase the degree of unfairness. This situation can be described through an example using the following workload: five jobs in group 1 execute for 10 ms after sleeping for 30 ms, while jobs in group 2 and group 3 execute for 1 ms after sleeping for 30 ms. The Share of each group is given by:

Share[1] = 50%,

Share[2] = 30% and

Share[3] = 20%.

Each time group 1 is selected to be the next group to run, the jobs in group 1 will consume CPU for a complete scheduling interval. When either group 2 or group 3 is selected to be the next group to run, the jobs in the group can only consume CPU for a very short period and go to sleep thereafter. This gives the invoking thread an opportunity to run. Therefore, the GCU of group 1 is greater than its group Share, and the GCUs of both group 2 and group 3 are lower than their group Share.

The invoking thread will signal the semaphore frequently when the selected group is either group 2 or group 3, because jobs in these groups are most likely to be in the sleeping state. If group 1 is selected to be the next group to run, the jobs in group 1 will receive more CPU share and the difference between its GCU and group Share will increase. While the jobs of group 1 are consuming CPU, the jobs in either group2 or group3 may become ready to run, but they cannot get any CPU resource immediately. Because group 1 always has enough jobs to consume the CPU, this will increase the unfairness of the scheduler.

To eliminate this unfairness, the scheduler needs to raise the priority of jobs in group 2 and group 3 to give them extra opportunities to run, even when group 1 is selected to be the next group to run. By doing this, the jobs in both group 2 and group 3 will immediately get opportunities to consume CPU when they become ready. The scheduler will also need to adjust the scheduling interval for the next scheduling.

The pseudo code of the main thread for the Dynamic-Priority Scheduler is shown in Figure 3-8.

```
Begin
    Initialization
    Create jobs for each group
    Initialize the semaphore
    Create the invoking thread
    {Begin scheduler}
    Do scheduling loop
        Lower priority value for the jobs of the running group
        Compute GCU for each group
        IF the selected group has consumed CPU within the scheduling interval
            Choose the group with the smallest ratio of GCU and Share as the next running group
            Raise the priority of the selected group
        ELSE (no job in the previously selected running group that is ready to run)
            Choose the group with a smallest ratio of GCU to Share from the remaining groups
           to be the next running group
           Raise the priority of the selected group
            IF the GCU of the chosen group is greater than its group Share
                Find the group with the lowest ratio between its GCU and Share
                Raise the priority of this group to the second highest value
            ENDIF
        ENDIF
        Wait on the semaphore
    Until the experiment time past
    {End scheduler}
    Compute GCU for each single group
    Printout the result
End
```

Figure 3-8 Pseudo Code of the Main Thread for the Dynamic-Priority Scheduler

3.4.4 Adaptive Scheduler

The Dynamic-Priority scheduler schedules more efficiently than the Static scheduler by distributing CPU resource more evenly among the groups. However, its

frequent scheduling can result in higher overheads. An Adaptive scheduler is introduced to combine the ideas underlying both the Static and Dynamic-Priority schedulers. The pseudo code of the main thread of the Adaptive scheduler is shown in Figure 3-9.

```
BEGIN
        Initialization
        {Begin scheduler}
        Do scheduling loop
                Set the priority value for the jobs of each group to lowest
                Update the GCU for each group
                Carry out a CPU share comparison among the three groups
                Select the group with the smallest ratio of GCU to Share
                Increase the priority of jobs in the selected group to the highest
                Carry out a CPU share comparison between the other two groups
                Select the group with a smaller ratio of GCU to Share
                Increase the priority of jobs in this selected group to the second highest
                {Adjust the scheduling time interval}
                IF the ratio of GCU to Share for the highest priority group is less than 0.9
                        Multiply the scheduling time interval by 2
                ELSE IF the ratio is greater than 1.1
                        Multiply the scheduling time interval by 0.5
                ENDIF
                Sleep for a duration equal to the adjusted scheduling interval
        Until the experiment finish
        {End scheduler}
        Compute the GCU for every single group
        Printout the result
END
```

Figure 3-9 Pseudo Code of the Main Thread for the Adaptive Scheduler

The Adaptive scheduler uses a variable scheduling interval that is calculated based on the ratio of GCU to Share for the next selected running group. Sometimes a group is selected to be the next group to run, but the GCU of jobs in this group is much

lower than the Share of the group. In order to increase the opportunities to run for the jobs in this group, the scheduling interval is increased when this group is selected to be the next group to run. Every time the scheduler wakes up, it performs the priority adjustment for all the jobs in every group. The priority of jobs in each group is based on the ratio of its GCU to Share.

3.5 Experimental Results

This section presents the performance of all the schedulers under various workloads. Various factors affect the scheduler's performance in different ways.

There is no throughput shown in the first two sections as total throughputs, for all the schedulers are the same. For the experiments described in those sections, the execution times and the sleep times for all the jobs are the same. No matter how the scheduler performs the scheduling, the total numbers of job cycles per second remain the same.

3.5.1 VGUs Match Shares Exactly

This test is expected to show the behavior of each scheduler when the VGU of each group is equal to its group Share. Because the jobs in all the groups continue to keep the CPU busy with no overload, regardless of the scheduling strategy, each group is expected to get the group Share it is entitled to. The input data for this test is shown in Table 3-4.

Given an exact match between the VGU of each group and the group's Share, the ADR for all the schedulers should be close to zero. The results achieved with each

scheduler are shown in Table 3-5. As expected, the ADRs for all the schedulers are close to zero, as all the groups obtained their share to consume the CPU resource.

Table 3-4 Input Data for Experiment in which VGUs Match Shares Exactly

	Group 1	Group 2	Group 3
No. of Jobs	5	4	3
Exec[i] (ms)	12	9	8
Slp[i] (ms)	108	111	112
VGU[i] (%)	50	30	20
Share[i] (%)	50	30	20

Table 3-5 Results when the VGUs Match Shares Exactly

	Equal-Priority	Static	Dynamic-Priority	Adaptive
GCU[i] [i = 1/2/3] (%)	50/30/20	49.7/29.8/20.0	49.9/30.0/20.0	49.4/29.6/19.7
Total group Utilization (%)	100	99.46	99.87	98.74

From Table 3-5 it is also evident that the Equal-Priority scheduler had the lowest overhead. This is because all the jobs consume CPU sequentially, following a cycle. Each job executes exactly once every 120 milliseconds. As illustrated in Figure 3-10, the vertical lines stand for the start/stop execution times of the jobs in each group. Staring with group 1, the first job (j11) finishes running after 12 ms, and sleeps for 108 ms; within its sleeping period, all other 11 jobs complete their runs. The jobs in group 1 finish their executions after 60 ms (5 * 12 ms), the jobs in group 2 are executed in the next 36 ms (4 * 9ms), and the jobs in group 3 use the next 24 ms (3 * 8ms). After 120

ms, the first job of group 1 wakes up and goes into the next execution cycle, and all the other jobs wake up sequentially.

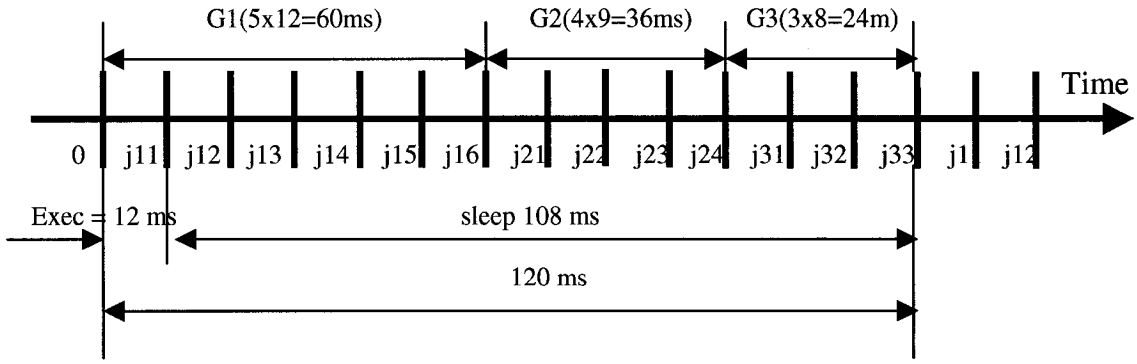


Figure 3-10 Job Execution Sequences Achieved with the Equal-Priority Scheduler

The Dynamic-Priority scheduler has lower overhead than both the Static and Adaptive schedulers, as it does not need to change the job execution sequence under this workload. The job execution sequence is the same as that in the Equal-Priority scheduler. At any time after the first round of job executions, which takes 120 ms, there is only one group of jobs ready to consume the CPU resource, and these jobs are given the highest priority.

With the Static and Adaptive schedulers, the scheduler may start scheduling in the middle of a job execution. This results in changes to the execution order of all the jobs, as some pre-emption will occur during the scheduled period. The Adaptive scheduler has higher overhead than the Static because it has more priority switches.

3.5.2 All Jobs with Equal Execution Times and Sleep Times

If all the jobs have equal execution and sleep times, as the number of jobs in each group is fixed, the ratio of their VGUs is fixed at VGU[1] : VGU[2] : VGU[3] = 5 : 4 : 3. The purpose of this test is to investigate the ADR and total group utilization of each

scheduler, giving all the groups of jobs the same execution/sleep time. The input data for this test is shown in Table 3-6.

Table 3-6 Input Data for Experiment with Equal Execution/Sleep Times

	Group 1	Group 2	Group 3
No. of Jobs	5	4	3
Exec[i] (ms)	varies	Same as group 1	Same as group 1
Slp[i] (ms)	30	30	30
VGU[i] (%)	varies	varies	varies
Share[i] (%)	50	30	20

Figure 3-11 shows the ADR of the schedulers for different execution times. The jobs in all the groups have the same execution time (Exec[1] = Exec[2] = Exec[3] = ExecutionTime), which is shown on the horizontal axis.

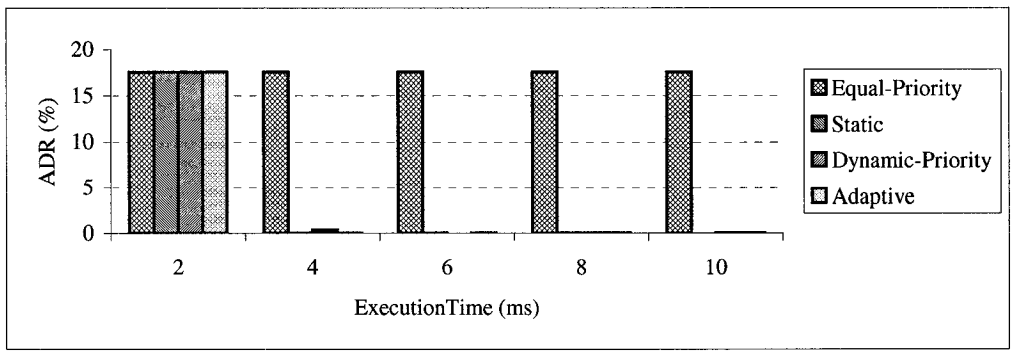


Figure 3-11 ADRs for Equal Execution/Sleep Times

As shown in Figure 3-11, when ExecutionTime is greater than or equal to 4 ms, the VGUs of all the groups are greater than their Shares. The three schedulers, Static, Dynamic-Priority and Adaptive, distribute the CPU resource fairly to each group, based on its Share. With the Equal-Priority scheduler, all the jobs run exactly the same number of times as they run under FCFS, so the GCU of each group is proportional to its VGU

(see Section 3.4.1). The ADR of the Equal-Priority scheduler is therefore fixed at around 18%, while the ADRs of the other three schedulers are close to zero. When each group's VGU drops below its group Share (ExecutionTime = 2 ms), there is not much work to keep the CPU continually busy. In this case, all the schedulers produce an ADR of 18%.

The number of times each scheduler is run is shown in Table 3-7. As shown in this table, the Dynamic-Priority scheduler runs more frequently and consumes more CPU time, as only one group is selected to run within a scheduling interval. The drop in utilization with the Dynamic-Priority scheduler is a little more than the other schedulers (see Figure 3-12).

Table 3-7 Number of Scheduling Operations with Equal Execution/Sleep Times

Exec[i]	Equal-Priority	Static	Dynamic-Priority	Adaptive
10	1	1483	2884	1505
8	1	1483	3725	1504
6	1	1483	6273	1510
4	1	1483	16001	1503

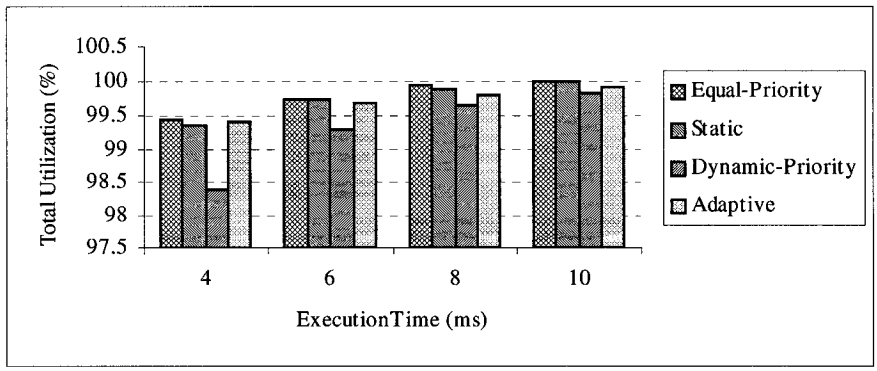


Figure 3-12 Total Group Utilization with Equal Execution/Sleep Times

3.5.3 Two Groups with VGUs Greater than Their Shares

When there are two groups for which the VGUs are greater than their Shares and the third group's VGU is smaller than its Share, the jobs of the third group cannot consume the group's entitled Share. This unused CPU share will be distributed between the first two groups. This test aims to examine each scheduler's ADR, utilization, and throughput in such a situation. The input data is shown in Table 3-8.

Table 3-8 Input Data for Experiment with VGUs of Two Groups Greater than Their Shares

	Group 1	Group 2	Group 3
No. of Jobs	5	4	3
Exec[I]	varies	Same as group 1	1 ms
Slp[i]	30 ms	30 ms	30 ms
VGU[i]	varies	varies	9.68%
Share[i]	50%	30%	20%

The ADRs of different schedulers are presented in Figure 3-13. The horizontal axis shows the execution time for the jobs in each group (Exec[1]//Exec[2]//Exec[3]) in ms, and the sleep time for each job is 30 ms.

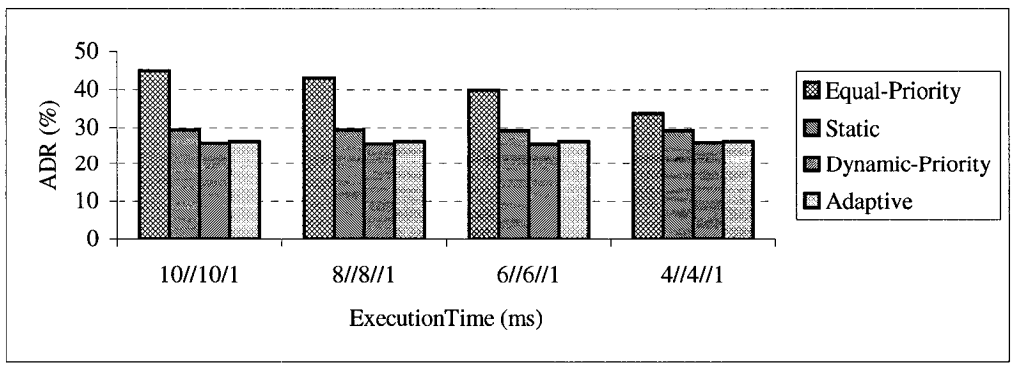


Figure 3-13 ADRs for Two Groups with VGUs Greater than Their Shares

Because the Equal-Priority scheduler gives each job an equal opportunity to consume CPU, every job receives the same number of chances to carry out an execution, and the GCU of each group is always proportional to its VGU. As the execution time for jobs within the first two groups decreases, as shown in Figure 3-13, the VGUs of these groups become closer to their Shares, and the ADR becomes smaller and smaller. Note that the smallest ADR achieved with any execution time setting and any scheduling policy is 25.7%. One of the reason for this is that the VGU of group 3 is much smaller than its Share. This increases the value of DR[3] for any scheduling policy. Since VGUs of group 1 and group 2 are larger than their Shares, the spare capacity is consumed by these groups. This leads to increases in the values of DR[1] and DR[2] as well.

Each of the other three schedulers achieves a certain level of improvement over Equal-Priority. Both the Dynamic-Priority and Adaptive schedulers perform better than the Static, as they provide more levels of priority.

The Static scheduler is the simplest and runs as follows. The jobs in group 3 are always given the highest priority, as the ratio of GCU[3] to Share[3] is smallest. The jobs in both group 1 and group 2 are put into the same lowest priority queue, because each of their GCUs is greater than its group Share. Because the jobs in group 3 sleep most of the time, group 1 and group 2's jobs get opportunities to consume CPU on an FCFS basis, which is similar to the Equal-Priority scheduler. As a result, the GCUs of the first two groups are proportional to their VGUs. Therefore the ADR for the Static scheduler is a little bit higher than that for both the Dynamic-Priority and the Adaptive schedulers (as shown in Figure 3-13).

With the Dynamic-Priority scheduler, at least one group of jobs will be given the highest priority each time. Because the jobs in group 3 have the shortest execution time and long sleep times, the jobs are in the sleep state most of the time. Therefore, the invoking thread has more opportunities to wake the scheduler up and bring about frequent scheduling. When one of the first two groups is selected to be the next group to run, it is most likely that its current GCU is already greater than its group Share. Because the GCU of group 3 is much smaller than the Share of group 3 (see Section 3.4.3), the jobs in group 3 have been set at the highest priority. This gives group 3 the maximum possible GCU, and the GCUs of the first two groups are proportional to their Shares.

The Adaptive scheduler runs with less complexity than the Dynamic-Priority scheduler while managing to achieve the same level of fairness (see Figure 3-13). The jobs in group 3 are always set to the highest priority, as its group VGU (9.5%) is much smaller than its group Share (20%). The priorities of the jobs in group 1 and group 2 switch between the medium and lowest levels for each scheduling interval, based on the comparison of their GCU and Share. The group with the smaller ratio of GCU to Share will be given the medium priority. As the jobs in group 3 are sleeping most of the time, the jobs in the group with medium priority will have precedence over the jobs in the group with the lowest priority.

Table 3-9 shows the total number of times the schedulers run during an experiment. The first column of the table presents the execution time for the three groups (Exec[1]/Exec[2]/Exec[3]). The other columns show the number of times each scheduler runs within the given experiment time.

Table 3-9 The Total Number of Runs for the Schedulers

Exec[i]	Equal-Priority	Static	Dynamic-Priority	Adaptive
10/10/1	1	1483	32208	13567
8/8/1	1	1483	40767	26595
6/6/1	1	1483	49738	21338
4/4/1	1	1483	98384	29875

Because there are frequent scheduler invocations, the Dynamic-Priority scheduler has more overhead than the Adaptive scheduler. As shown in Figure 3-14, the Dynamic-Priority scheduler has the lowest system utilization.

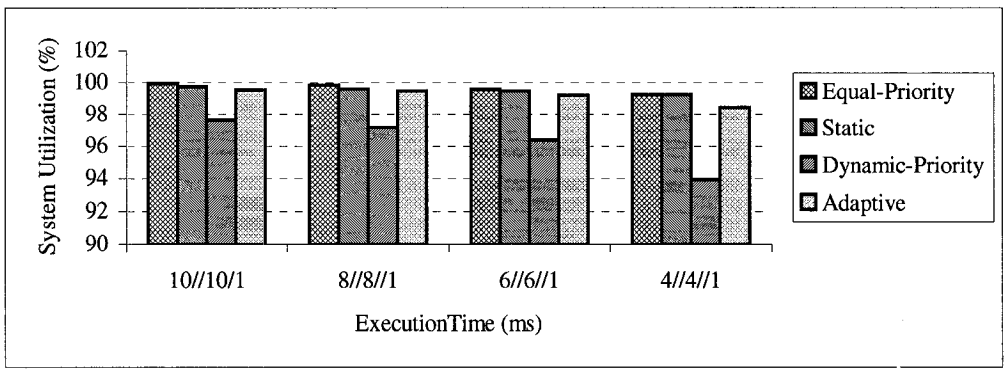


Figure 3-14 System Utilization for Two Groups with VGUs Greater than Their Shares

The throughput of each scheduler is shown in Figure 3-15. As shown in this figure, the Equal-Priority scheduler has the smallest throughput as the execution of the jobs in group 3 are shorter and less frequent than those in the other three schedulers. The Static scheduler has the highest throughput as it gives rise to a lower number of preemptions, compared to the Adaptive scheduler. The Adaptive scheduler has more preemptions than the Static, but fewer than the Dynamic-Priority scheduler. As a result, the Adaptive scheduler gives a higher throughput with fair share.

Figure 3-15 Throughput for Two Groups with VGUs Greater than Their Shares

Looking at Figures 3-13, 3-14 and 3-15 simultaneously, the Adaptive scheduler seems to be an attractive choice when all the metrics are considered. The Adaptive scheduler gives the maximum possible CPU share to each group according to their Shares, while the Dynamic-Priority scheduler uses more CPU resource to perform the scheduling.

3.5.4 VGU of One Group Greater than its Share

This section analyzes the ADRs of the different schedulers when there is only one group in which the VGU is greater than its Share, and the VGUs of the other two groups are much smaller than their Shares. Table 3-10 shows the test input data.

Table 3-10 Input Data for Experiment with VGU of One Group Greater than its Share

	Group 1	Group 2	Group 3
No. of Jobs	5	4	3
Exec[I]	varies	1 ms	1 ms
Slp[i]	30 ms	30 ms	30 ms
VGU[i]	varies	12.9%	9.68%
Share[i]	50%	30%	20%

The ADRs for the different schedulers are presented in Figure 3-16. The results shown are similar to those presented in the previous section. The system behavior is discussed briefly.

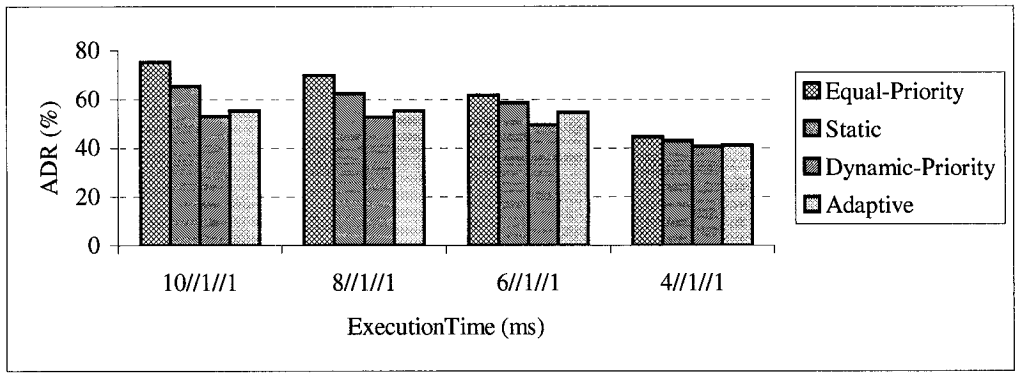


Figure 3-16 ADRs when the VGU of One Group is Greater than its Share

With the Static scheduler, the jobs in group 1 are set at the lowest priority all the time, as GCU[1] is greater than Share[1]. The jobs in both group 2 and group 3 are switched between the highest and medium priorities as the GCUs of both groups are much smaller than their Shares. Based on a comparison of CPU usages, the group with the smaller ratio of GCU to Share will be given the highest priority. In one specific scheduling period, only one group is selected. Jobs in the other group are put in the same queue as the jobs in group 1. Once the jobs in the scheduled group are in the sleeping state, the jobs in the other group share the CPU with the jobs in group 1 on an FCFS basis, and the CPU resource is distributed in a ratio that is proportional to their VGUs.

For the Dynamic-Priority scheduler, the invoking thread has more chance of waking up and signaling the scheduler when either group 2 or group 3 is selected to be the running group, because the jobs in both groups are in the sleeping state most of the

time. As described in Section 3.4.3, the jobs in the last two groups are set at a higher priority most often. This is because the GCUs of the last two groups are lesser than their Shares while the GCU of the first group is higher than its Share.

For the Adaptive scheduler, the jobs in group 1 are always set at the lowest priority, as GCU[1] is greater than Share[1] almost all of the time. The jobs in group 2 and group 3 switch priority between the medium and highest, based on the difference between their GCUs and Shares. Because the jobs in both group 2 and group 3 are set at a higher priority than the jobs in group 1, they will immediately use their allocated CPU resource once they are ready.

Figure 3-17 shows the utilization of group 1 for each scheduler. The utilization of group 2 is shown in Figure 3-18, while Figure 3-19 shows the utilization of group 3. These three figures show that both Dynamic-Priority and Adaptive schedulers enable the last two groups to be utilized in a way which is very close to their VGU values, but the Adaptive scheduler enables group 1 to achieve a higher utilization than the Dynamic-Priority scheduler does.

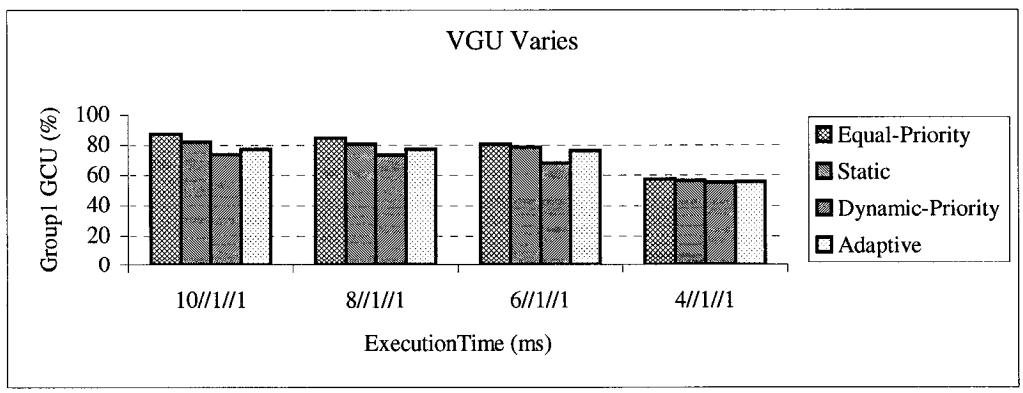


Figure 3-17 Group1 CPU Utilization

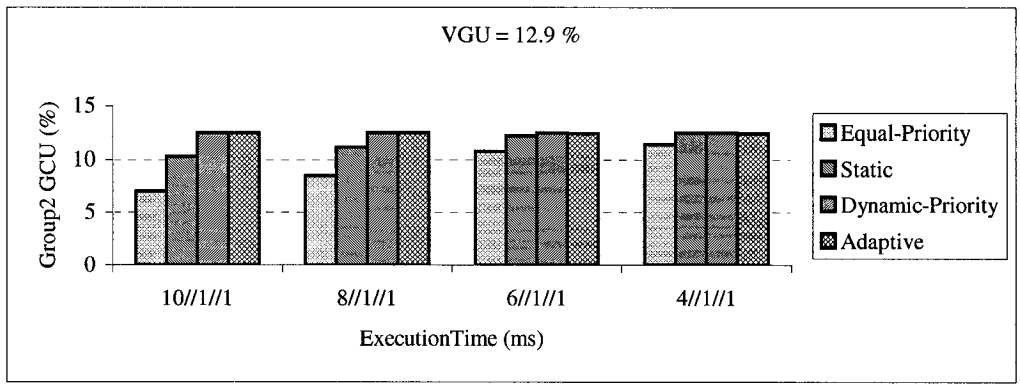


Figure 3-18 Group2 CPU Utilization

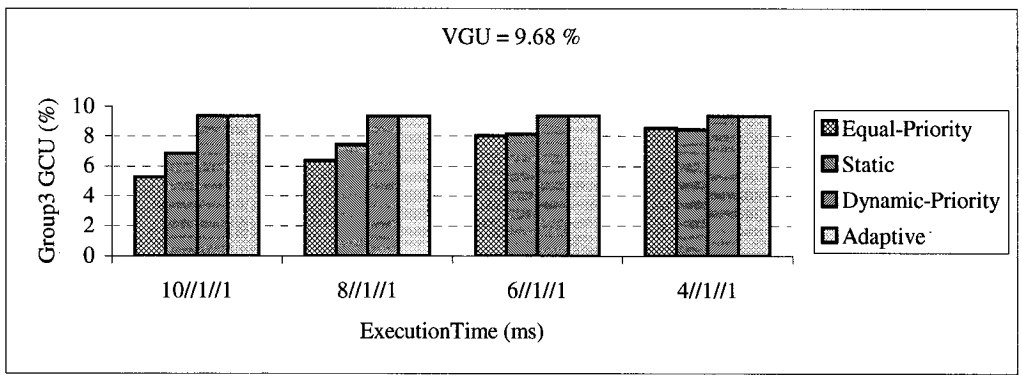


Figure 3-19 Group3 CPU Utilization

The overall throughput of all the schedulers is shown in Figure 3-20. As the Adaptive scheduler gives the short jobs in both group 2 and group 3 more opportunities to consume CPU resource, and runs the scheduler a lower number of times, it achieves the highest throughput overall.

To conclude from the results shown in Figures 3-16 to 3-20, the Adaptive scheduler performs the best in terms of CPU utilization and system throughput overall. Compared to the Dynamic-Priority scheduler, although the ADRs of the Adaptive

scheduler are slightly higher, it produces the highest group utilizations for all the groups at the same time.

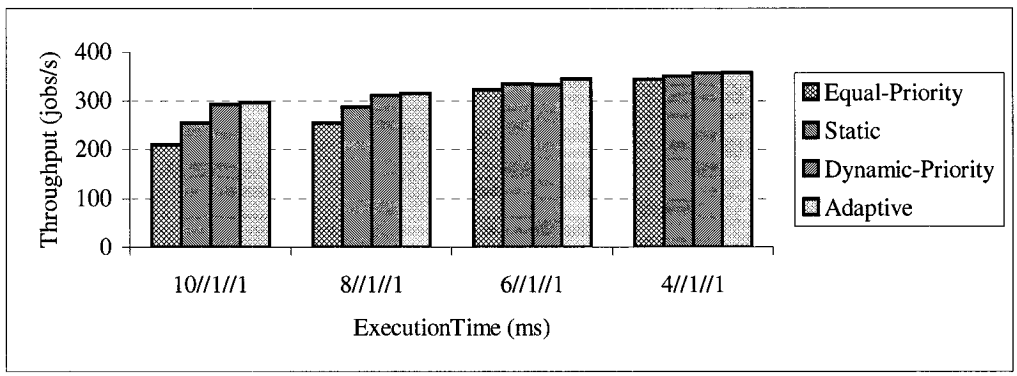


Figure 3-20 Throughput when the VGU of One Group is Greater than its Share

3.5.5 VGUs of All Groups Greater than Their Shares

When all the VGUs are greater than their Shares, the utilization of each single group should match its own share. The ADR is therefore expected to be close to zero. The aim of the experiments described in this section is to test the schedulers with various workloads in order to understand their behavior. The input data is presented in Table 3-11.

Table 3-11 Input Data for Experiment with All VGUs are Greater than Their Shares

	Group 1	Group 2	Group 3
No. of Jobs	5	4	3
Exec[i] (ms)	varies	varies	Varies
Slp[i] (ms)	30	30	30
VGU[i] (%)	>50	>30	>20
Share[i] (%)	50	30	20

Figure 3-21 shows that all the schedulers achieve this goal, except the Equal-Priority scheduler; for this scheduler, the GCU of one group is proportional to its VGU (refer to Section 3.5.2).

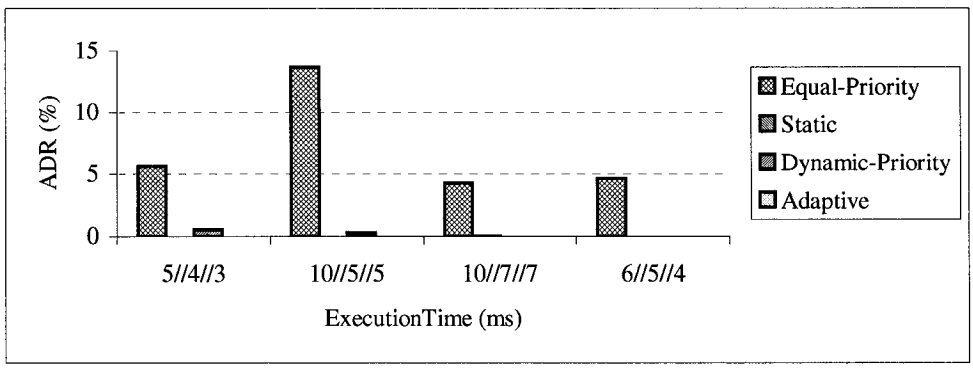


Figure 3-21 ADRs when All the VGUs are Greater than Their Shares

Figure 3-22 shows the system utilization with each scheduler. Due to more frequent scheduling, the utilization achieved with the Dynamic-Priority scheduler is slightly lower than the others, which achieved utilizations close to 100%.

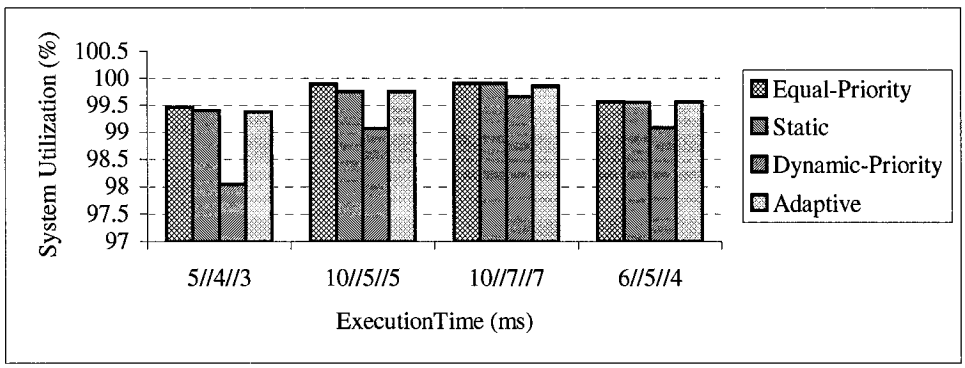


Figure 3-22 System Utilization when All VGUs are Greater than Their Shares

3.5.6 Relationship between Base Scheduling Interval and Overhead

The Base Scheduling Interval determines the running frequency of the scheduler when the VGUs of all groups are greater than their shares. With the Static scheduler, when the BSI becomes smaller, the scheduler runs more frequently. If the VGUs of all the groups are similar to their Shares, the Dynamic-Priority and the Adaptive schedulers will run for a smaller number of times. Therefore the input data used in Section 3.5.4 was selected for this test, as the VGUs are very different from their Shares. Figure 3-23 shows the overheads of different schedulers for various base scheduling intervals, ranging from 10 ms to 1000 ms. Although the overhead of the Static scheduler tends to become higher with a decrease in BSI, the difference is marginal. With the Adaptive scheduler, the differences in the overheads achieved with different BSIs are very limited. BSI has very little effect on the Dynamic-Priority scheduler overhead, as it wakes up within much shorter intervals, in most cases. Overall, for the BSI values experimented with, BSI seems to have a small impact on overhead.

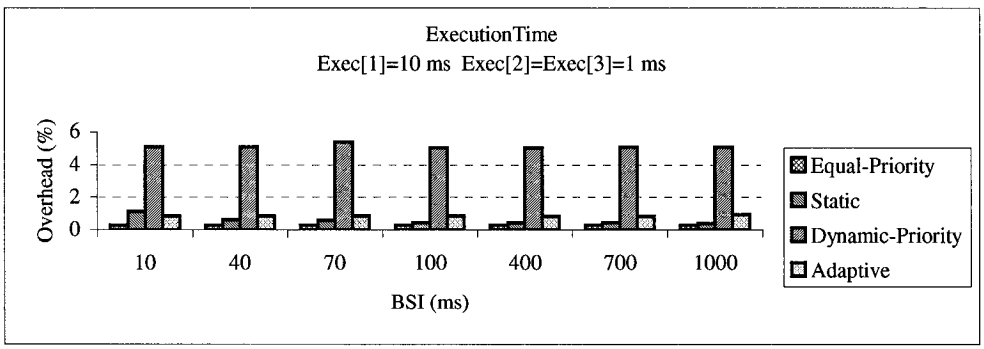


Figure 3-23 The Effect of Base Scheduling Interval on Scheduling Overhead

As the above figure shows, because it runs frequently, the Dynamic-Priority scheduler has the highest overhead.

3.6 Summary

In order to devise a Fair-Share scheduler, this thesis investigated four priority-based scheduling policies. Based on the results presented in the last section, a summary of the characteristics of these schedulers follows:

- The Equal-Priority Scheduler gives a group the ability to increase its CPU usage. A group can get more CPU share by creating extra jobs. It is not fair to each group unless each group's VGU is exactly the same as its Share.
- The Static Scheduler can achieve Fair-Share only when each group's maximum utilization is larger than or equal to its Share.
- The Dynamic-Priority Scheduler distributes CPU resource to each group in a way that is proportional to its Share. It produces a relatively higher overhead in comparison to other schedulers, which therefore reduces the overall useful system utilization.
- In most cases, the Adaptive Scheduler performed the best in terms of Fair-Share scheduling and overall useful system utilization. In the event that the VGU of one group is smaller than its predefined CPU share, the Adaptive scheduler provides the maximum share achieved for this group while maximizing the other group's CPU utilization.

A discussion of the important characteristics of the Adaptive scheduler is presented next:

- Each group of users gets its entitled fair share of CPU when all the groups are ready to consume the CPU resource.

- If the CPU utilization of one group is greater than its Share, it should get less opportunity to use the CPU resource. When the CPU utilization of a group is smaller than its entitled Share, it should be given more chance to consume the CPU resource.
- Within a specific scheduling period, if one group is not ready to use its CPU shares, the CPU resource is granted to the active jobs in the other groups.

Chapter 4 Case Study: Network Router

This chapter describes the investigation of resource management strategies for a network of routers. The system and its performance prototype are introduced in Section 4.1. The performance metrics are given in Section 4.2. Section 4.3 and Section 4.4 introduce two types of resource management strategies: the packet dropping policies and the scheduling policies, respectively. The implementation details are presented in Section 4.5. Section 4.6 provides the experimental results, which is followed by a summary in Section 4.7.

4.1 Environment Settings

A ChorusOS-based performance prototype of a network router is constructed. This prototype contains only the components that are necessary for investigating the performances of the resource management strategies. This performance prototype consists of three PCs and one Solaris workstation. These three PCs are connected to each other through an edge switch, while the PCs connect to the Solaris workstation through a private network. The workstation is the host, on which the ChorusOS development environment has been completely installed. The three PCs are the targets on which the ChorusOS runtime environment is installed. The applications running on the targets are remotely controlled by the host. Each of these three targets acts as a network router.

The performance prototype of network router is based on the CG-Net system [16], which is a software-based network router system developed by Nortel Networks. It is

designed for the performance improvement study of an IP router for different routing and forwarding protocols, including OSPF and MPLS.

4.1.1 Description of CG-Net

CG-Net consists of a network of software-based routers. Each router is composed of multiple processes. Two routers communicate through the UDP or the TCP protocol. The complete process view of the routing system is shown in Figure 4-1. The Command Line User Interface (CLUI) controller is unique to all three routers.

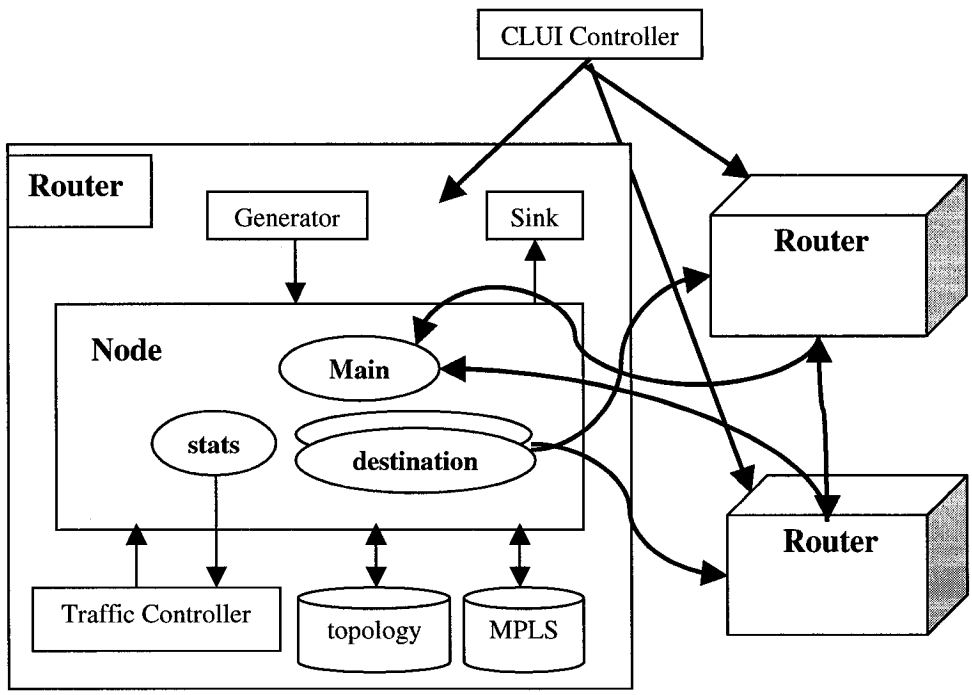


Figure 4-1 CG-Net Structure

The Generator process generates packets and sends them to the Node process. The Node process receives packets from the Generator or other Node processes, and

handles the packets. The Sink process only consumes packets. The Traffic Controller dynamically adjusts the traffic based on the link utilization. Each process is described in more detail.

- *Generator:* Generator is a single-threaded process. It generates packets at a specified rate and puts the packets into a message queue. The rate is configurable. The Node process will pick up the packets from the queue and process them.
- *Node:* Node is a multi-threaded process and the main process of the router. Once it has received a packet from the message queue, it will examine the packet header and retrieve the necessary information from the database or routing table, and forward the packet to its destination. The destination could be either a neighboring router or the *sink* process that is directly connected to it. It has at least the following four threads.

  Main: the main thread obtains a packet from the message queue that is updated by the *Generator* process, retrieves the relevant information, undertakes a computation and finds the next destination. It puts the packet into the message queues of either the destination threads or the *sink* process. The insertion is based on the type of the packet. The command packet will be inserted at the front of the queue, while the data packet will be inserted at the rear of the queue.

  Destinations: A destination thread obtains a packet from the main thread and sends it to the destination through a UDP or a TCP protocol. At least two destinations are required for the performance prototype as the packets

generated need to be distinguished from different sources or sent to different destinations.

Stats: The statistics thread gathers the packet-processing statistics information of the router in order to adjust the routing policy.

- *Sink:* The sink process is a single-threaded process. It receives packets from a destination thread of the node process. The sink process emulates the end user or an edge switch.
- *CLUI Controller:* The CLUI controller is a single-threaded process. It can be used to change the configurations and parameters to adjust the network topology and properties through a command line user interface.
- *Traffic Controller:* The traffic controller is a single-threaded process. It retrieves the statistics information from the stats thread of the Node and adjusts the rerouting strategy of the packets accordingly. This process is used primarily for network traffic engineering. Once the router is in the steady state, it is rarely active.

There are also two data stores that reside within the router. One is related to packet routing; the other is related to packet forwarding.

- *Topology data store:* The topology data store stores the routing information used by OSPF.
- *MPLS data store:* The MPLS data store stores the forwarding path information used by the MPLS protocol.

These active data stores are updated dynamically to maintain the current topology of the network. This study focuses on three processes: *Generator*, *Node*, and *Sink*.

4.1.2 Performance Prototype of CG-Net

Network resources are limited for every Internet Service Provider (ISP). Each ISP distributes its limited network resources to its customers, based on their service criteria. The most important customers will usually get more network resources at a given time. The ordinary customers have fewer resources. Nonetheless, the ISP does have to provide resources to ordinary customers at certain time periods. Different priority levels can be associated with different classes of customers.

For this prototype, we are concerned with three packets groups, each of which is associated with a different priority. The highest priority packets correspond to the *Gold* service class and are generated by the most important customers. Those packets are put into the highest priority queue. The middle priority packets correspond to the *Silver* service class and they go into the middle priority queue. The lowest priority packets correspond to the *Bronze* service class and are generated by the ordinary customers. These packets go to the lowest priority queue. The structure of the simulated routing system is shown in Figure 4-2.

There are three processes within each router: Generator, Sink and Node. The generator and sink processes are the same as those described in the previous section. They connect to the node through message queues. The node process consists of the following threads: Receiver1 receives the packet from the generator process; Receiver2 receives the packets from the other routers. There are three processQ threads, each of

which processes one packet group with a specific priority; and each of two destination threads sends the packets out to their destination router. Message queues sit between the Receiver1 thread and each of the three processQ threads, and also between the processQ thread and each of the two destination threads.

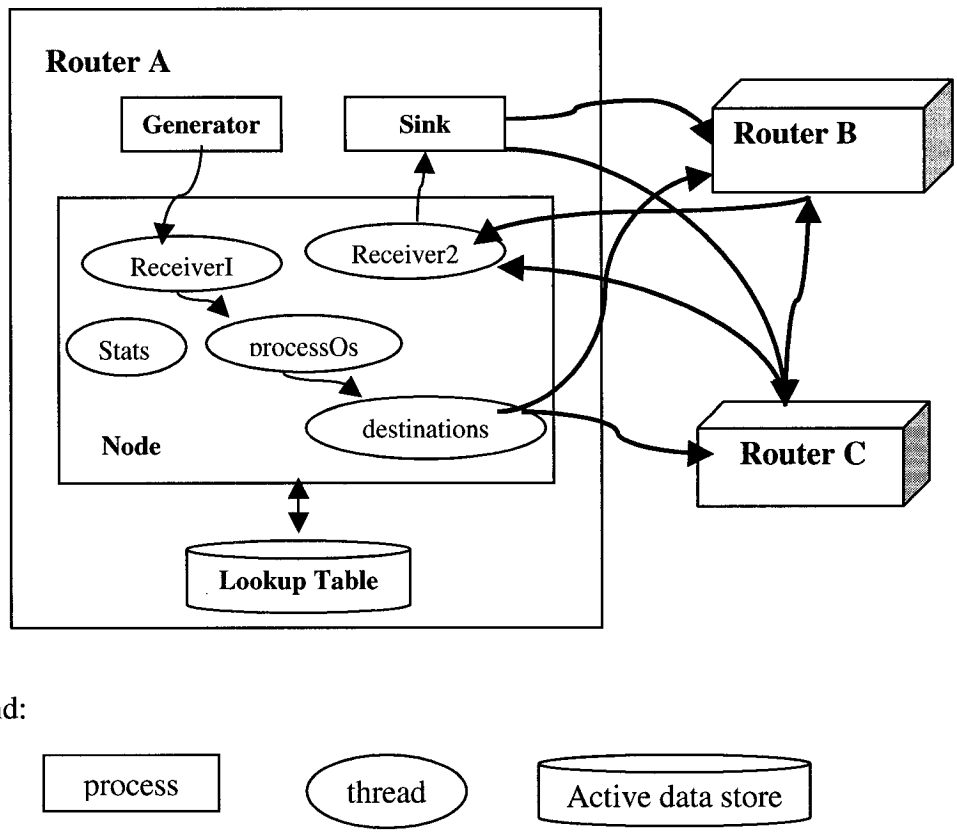


Figure 4-2 Structure of Performance Prototype of the CG-Net

Assuming that the router uses a constant amount of time to process a packet in each of the three message queues, a share of the CPU resource will be given to each of the priority queues to process the packets stored in each queue.

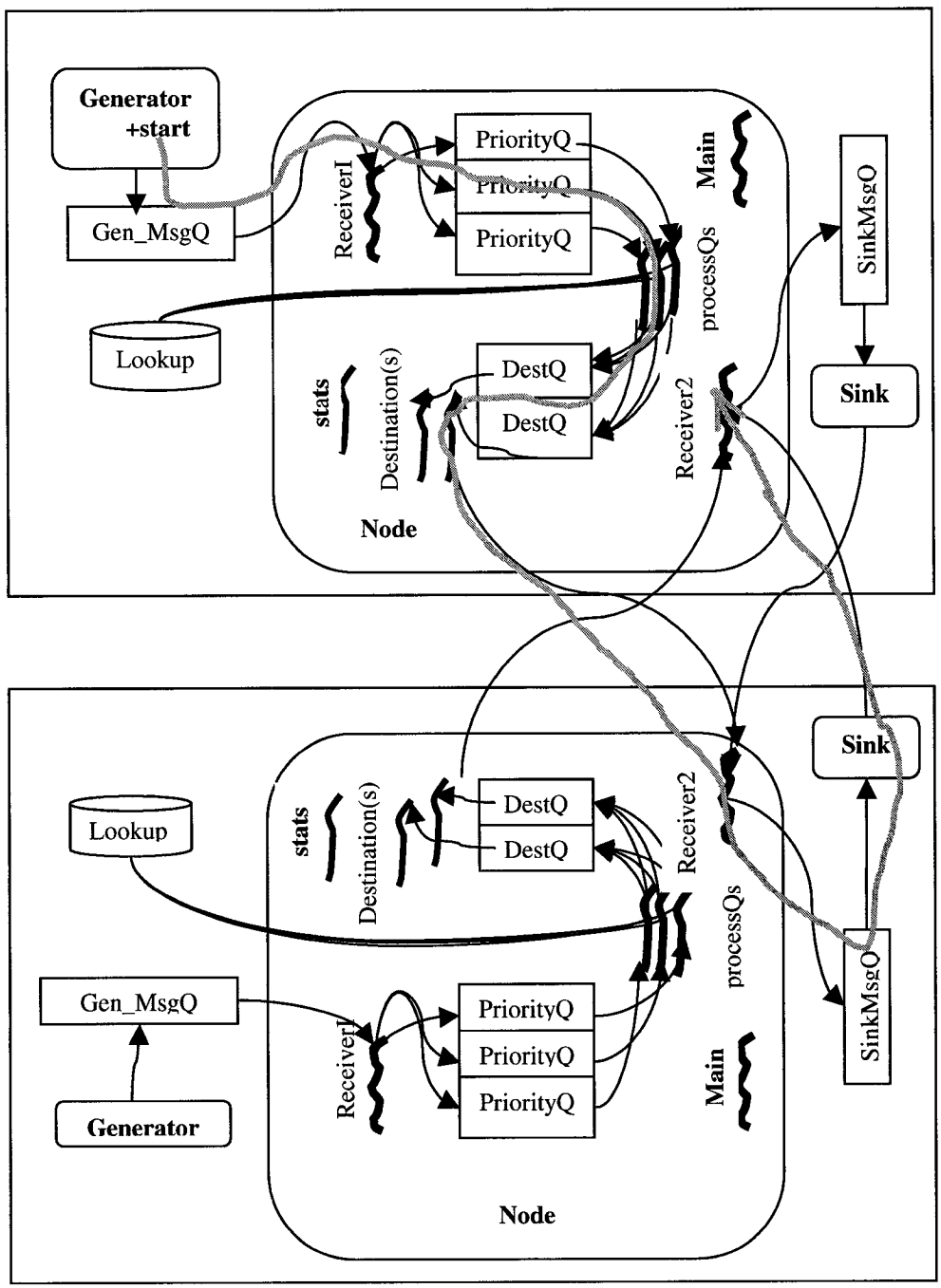
Because the ChorusOS provides a memory-sharing mechanism, namely Message Space, it is convenient for different processes to share the same message address through reference passing in real time.

The pure priority-based packet-processing system may not be suitable for the lowest priority packets. If the number of higher priority packets becomes large enough, there may be a starvation problem for the lower priority packets in the message queue. To overcome this problem, this study implements multiple queues for different packet groups, one queue per packet group. Each priority queue will have a different number of packets during execution. The scheduler needs to ensure that the packets in each queue have a chance to be sent out, sooner or later.

The message flows between the nodes are shown in Figure 4-3. A packet generated by the *Generator* process is sent to the generator message queue (Gen-MsgQ), which is shared with the *Node* process.

Within the *Node* process, the Receiver1 thread retrieves the packet from Gen-MsgQ, extracts its priority information, and puts it into the corresponding priority queue (PriorityQ). The packet is then picked up by one of the processQ threads for further processing. After processing, this processQ thread puts the packet into the appropriate destination queue (DestQ). The two DestQs correspond to the connections to other routers in the system. After that, one of the destination threads will read the packet and send it to the destination router.

The Receiver2 thread in the destination router will read the packet. The packet will later be sent to the Sink message queue (SinkQ). The Sink process will consume this packet and send an acknowledgement back to the Receiver2 thread of the source router.



Legend:

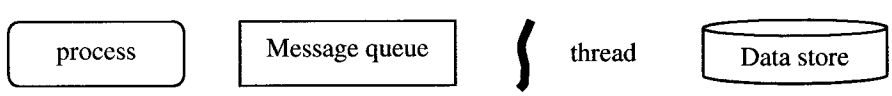


Figure 4-3 Message Flow Between Two Nodes

Within the *Node* process of the source router, the Receiver2 thread will compute the round trip time used for this packet and destroy the packet.

The long, curved gray line in Figure 4-3 illustrates the path a packet takes during its lifetime.

4.2 Performance Metrics

Many metrics have been used in network performance analysis and measurement. This thesis adopts three of the most common in order to measure system performance: packet drop rate, mean processing time and mean roundtrip time.

- Packet Drop Rate (%)

  This is statistical information for every router. Each link has a specified capacity and packets are dropped when the link capacity is exceeded. The packet drop rate reflects the capacity limitations of the network. Packet drop rate is defined as:

  PacketDropRate = (No. of PacketsGenerated – No. of PacketsSent) / No. of PacketsGenerated * 100%

  The Packet Drop Rate can be measured for every single packet group or for the overall router. For example, if the *Generator* process generates 1000 packets per second within a router and all the destination threads send out only 800 packets per second, the overall drop rate for the router will be 20%.

  A packet can be dropped from the priority queue when the queue length exceeds the limit. Because the buffer size for each queue within the router is limited, extra packets cannot be stored in the queue when the buffer is full. A packet can also be

dropped when the destination thread tries to send the packet out to the other router, but there is no bandwidth available.

- Mean Processing Time

  The mean processing time is intended to measure the time for which a packet stays in the router, i.e. from the time the packet arrives at the *Node* process to the time it leaves the *Node* process. This is the duration of time measured from the time the *Main* thread receives the packet to the time the *destination* thread sends out the packet.

- Mean Roundtrip Time

  The roundtrip time for a packet is the sum of the times used by the packet to travel from the source router to the destination router and the time for the acknowledgement from the destination to reach the source. This is the duration measured from the time the source *Generator* process generates the packet to the time at which the source *Node* process receives the acknowledgement. It is used to study the impact of network traffic and scheduling policies on performance.

4.3 Packet Dropping Policies

Because of the hardware limitations, there is a transmission limit for each of the links of the routers connected to each other over the network. The limitation affects the way ISPs provide services to users. In order for the ISPs to efficiently distribute their limited resources to different types of packets, a policy for packet dropping must be determined. Three distribution methods are described in this section.

4.3.1 Default-Link

With this dropping policy, all the packets arriving have the same priority to use the resource. They share the full capacity of a link between any two routers. Packets are processed on a First Come First Served (FCFS) basis.

When it retrieves a packet from the destination queue, the destination thread will check whether there is enough bandwidth to send it to the destination. If the available bandwidth is larger than the size of the packet, the *Node* will process it. However, irrespective of the customer class, if the available bandwidth is not large enough, the *Node* will drop the packet right away.

Obviously, the Default-Link dropping policy does not inherently provide any kind of differentiated service based on priority.

4.3.2 Fixed-Link

With a Fixed-Link policy, the capacity of each link is split into a fixed number of portions for the packet groups. The packet transmission decision is based on the available bandwidth of its group. That is, if there is not enough bandwidth available for its group to transmit this packet, instead of checking the available bandwidth for the overall link, the destination thread will simply drop it.

The Fixed-Link policy can provide differentiated service to different classes of customers. A higher portion of link capacity is given to higher priority packets, and lower priority packets are provided with a lesser amount of link capacity.

4.3.3 Dynamic-Link

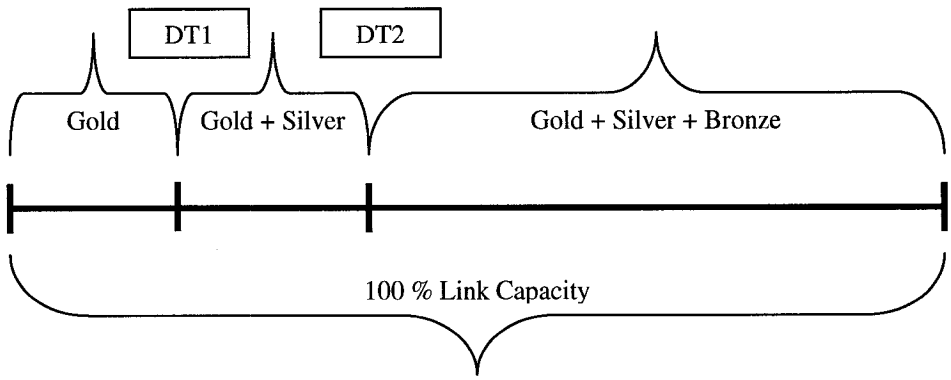
This policy is a combination of the above two policies. The principle of Dynamic-Link policy is to give only a fixed but relatively small portion of link capacity to the highest priority packets, and leave the rest to be shared by all the groups.

This concept comes from the following observation. With the Fixed-Link dropping policy, the link capacity is split into three portions: 50% goes to the Gold group, 30% to the Silver group and 20% to the Bronze group, for example. If the packets generated in the Gold group can use only 20% of their allocated total link capacity, the other 30% that is allocated to the Gold group will be wasted. With the Dynamic-Link policy, the other groups will share the unused portion of the link capacity.

A drop threshold is given for each of the lower priority packet groups. The value DT1 is for the Silver group and DT2 is for the Bronze group. As long as the available bandwidth is lower than the drop threshold DT2, the packet in the Bronze group will be dropped. Similarly, when the available bandwidth is less than the drop threshold DT1, packets in both the Silver and Bronze groups will be dropped. A simple illustration with pseudo code is shown in Figure 4-4. The total link capacity is divided into three portions: the first portion is reserved for Gold, the second portion is reserved for both Gold and Silver, and the third portion is used for all the groups.

For example, assume that the length of the incoming packet is 100 bytes and DT1 = 200 bytes and DT2 = 300 bytes. Based on the priority of this packet, the decision regarding whether to drop a packet is presented in Table 4-1.

As is evident from Table 4-1, because the entire link capacity is shared by three groups of packets, the Default-Link is actually a special case of the Dynamic-Link, where the threshold values are set to zero.




```
IF available BW is lesser than DT2 and packet belongs to the Bronze group
        Drop the packet
ELSE IF available BW is lesser than DT1 and packet does not belong to the Gold group
        Drop the packet
ELSE IF available BW is lesser than the packet length
        Drop the packet
ELSE
        Send the packet
END IF
```

Figure 4-4 Illustration of Link Sharing for the Dynamic-Link Strategy

Table 4-1 Packet Dropping with the Dynamic-Link Strategy

Available BW	Gold packet	Silver packet	Bronze packet
100	Send	Drop	Drop
> 200	Send	Send	Drop
> 300	Send	Send	Send

4.4 Scheduling Policies

To differentiate between different types of packets, the network router needs an algorithm to serve each group of packets at a different priority. A scheduler is required

for this purpose when multiple processes are running concurrently. Four scheduling policies are introduced in this section. Of these four scheduling policies, both Dynamic-Priority and Adaptive do not take packet priority into account. In these two scheduling policies, the scheduler makes the scheduling decision based on the predefined CPU share for each group and its actual CPU usage. Section 4.5.4 illustrates these policies in detail.

4.4.1 Equal-Priority

With this scheduling policy, all the processQ threads have the same priority, and the packets within each message queue are processed on an FCFS basis.

Packets processed with the Equal-Priority scheduling policy should not have any differences. This means that a similar drop rate and a mean processing time are expected to occur for all the packet groups. No matter how frequently groups of packets arrive, they are served in the same manner.

4.4.2 Fixed-Priority

In order to give each packet group a different priority, we devised the Fixed-Priority scheduler. This type of scheduler ensures that Gold packets will always be processed as quickly as possible, Silver packets will be processed if there are no Gold packets waiting, and Bronze packets will only be processed when there are no Gold or Silver packets. It is equivalent to the Priority Queuing strategy described in [4].

4.4.3 Dynamic-Priority

This is derived from the Dynamic-Priority scheduler described in the previous Chapter (see Section 3.4.3). It operates in the same way as the Dynamic-Priority scheduler, except for the number of threads the scheduler has to manage .

4.4.4 Adaptive

This is the same version as the Adaptive Scheduler that was developed in the last chapter. Initially all the processQ threads have the same priority as in the Fixed-Priority scheduler.

The scheduler wakes up periodically and checks the time used for the processQ thread for each packet group. It will set the processQ thread of the group with the smallest ratio of actual CPU share to predefined CPU Share to the highest priority. The processQ thread of the group that has the largest ratio will be set to the lowest priority. The processQ thread for the third group is set to medium priority. The scheduler will adjust the priority of each processQ thread according to its consumed CPU share.

4.5 Implementation Details

This section presents detailed discussions on each of the scheduling policies described in the previous section. The scheduling policies are simulated on ChorusOS. The performance prototype involves several parameters related to network and computer behaviors and performance. Those parameters will be discussed in Section 4.5.1. Two critical software components used in the prototype – the generator process and the processQs thread – will be described in Section 4.5.2 and Section 4.5.3, respectively. Detailed algorithms for the scheduling policies are presented in Section 4.5.4.

In addition, the prototype makes use of the message queue facility provided by ChorusOS for IPCs. Message queues are used in this prototype because the system calls are real-time compliant and messages are exchanged through a zero-copy interface.

4.5.1 Description of Parameters

The prototype usually involves various parameters to describe the environment. In this thesis, the following parameters are used.

Link Capacity (MBits/second)

This parameter refers to the limitation of packets flowing within the link between two routers. Once the total size of the incoming packets exceeds the link capacity, some of the packets will be dropped from the router.

Maximum Packet Waiting Time (μs)

This parameter simulates the hard limit for each packet to stay in the packet queue. If a packet waits in the priority queue for a time period exceeding this limit, the packet will be dropped immediately.

Mean Inter-arrival time (μs)

This parameter refers to the mean time interval between the arrival of two consecutive packets. The generator process will use this parameter to generate exponentially distributed packet inter-arrival times.

Share [i] [i = 1,2,3]

This parameter is the ratio of CPU distribution among different priority packets. For both Dynamic-Priority and Adaptive scheduling policies, the scheduler uses this parameter to make the scheduling decision. Ideally, if the CPU Share for the

Gold packet group is 50%, the scheduler will give the processQ thread that processes Gold packets about 50% of total CPU time.

Packet Group Ratio (p1:p2: p3)

This parameter specifies the ratio of packets generated in each group. It is used in the *Generator* process to generate packets with different priorities. For a long run, with every total number of (p1 + p2 + p3) packets, the *Generator* process will associate the highest priority, Gold, to p1 packets, the lowest priority, Bronze, to p3, and the rest of the p2 packets will be associated with the medium priority, Silver.

Packet Group Proportions [i] [i = 1,2,3]

This parameter specifies the proportion of packets generated for each group. Packet Group Proportion is related to the Packet Group Ratio as follows:

Packet Group Proportion [i] = pi/(p1+p2+p3)*100% where (i = 1,2, 3).

Link-Share[i] [i = 1,2,3]

This parameter specifies the proportion of the link capacity for each packet group. With a fixed link capacity, *Link-Share[1]* of the total link capacity will be given to the Gold packet group; *Link-Share[3]* of the total link capacity will be given to the Bronze packet group; and the rest will go to the Silver packet group.

Packet Processing Time(μs)

This parameter specifies the time that each processQ thread uses to process a packet. For each experiment it is held at a fixed value.

4.5.2 Packet Generation

The *Generator* process generates packets with exponentially distributed mean inter-arrival times. Each packet consists of several pieces of information. The packet format is *type*, *source*, *destination*, *length*, *priority* and *generation-time*. Each field is explained further as follows:

type

This is defined for packet processing. Originally, CG-Net had two types of packets: a command packet and a data packet [16]. The command packet has the high priority and will be processed immediately upon arrival, while the data packet is processed on an FCFS base. This prototype is only concerned with data packets. Therefore, all the packets have the same type.

source

This is the router where the packet is created.

destination

This is the last router to which the packet is forwarded.

length

This refers to the length of the packet, which varies from 64 bytes to 1500 bytes. For the sake of simplicity, we set the length to equal 1000 bytes.

priority

This corresponds to a level of service (Gold, Silver or Bronze) with which the packet will be provided.

generation-time

This refers to the time the packet is generated. The time is purely used for the performance measurement. It is used as the start time for the measurement of the roundtrip time for each packet. Because the *Generator* process and the main thread of the *Node* process have the highest priority, there is almost no delay from the time a packet is generated to the time the main thread gets it.

4.5.3 Packet Processing Threads

Each processQ thread in the *Node* process shown in Figure 4-2 handles the packets with the same priority. As the packet processing within the same queue uses the same procedure, each packet with the same priority requires the same amount of time to be processed within the *Node* process. Each packet has a hard limit of waiting time (*Maximum Packet Waiting Time*) to simulate the maximum delay. The limit is configurable. That means that if a packet waits in the queue for a period of time longer than the *Maximum Packet Waiting Time*, the packet will be dropped by the processQ thread immediately after being picked up from the queue.

4.5.4 Scheduler Implementation

This section describes the implementation details of the four different scheduling policies introduced in Section 4.4. As described in Chapter 3, the smaller the priority value, the higher the priority.

4.5.4.1 Equal-Priority

With Equal-Priority policy, all the processQ threads have the same priority. The complete priority settings for all the threads are shown in Figure 4-5. The *Generator*

process has the highest priority, to ensure that it will generate the packet at a specific time. Because all the groups of packets are processed at the same priority, there is no difference between Gold, Silver and Bronze packets.

Generator process	140
Main thread (scheduler)	142
Receiver1	146
Sink process	148
Receiver2	150
Destinations	150
processQ threads	155

Figure 4-5 Priority Setting of Equal-Priority Scheduling Policy

4.5.4.2 Fixed-Priority

This policy gives the highest priority to thread processQ1, which processes the highest priority Gold packets, and the lowest priority to thread processQ3, which processes the Bronze packets. ProcessQ2 is given medium priority because it processes the Silver packets. The priority settings for the processQ threads are as follows:

- processQ1: 152
- processQ2: 155
- processQ3: 158

The priorities of the other processes and threads are the same as those in Equal-Priority. It is expected that the highest priority Gold packet will always be processed using the shortest time, while the lowest priority Bronze packet will remain within the router the longest, waiting for processing.

4.5.4.3 Dynamic-Priority

The initial setting of the priority is the same as that of the Fixed Priority. What is different with this scheduler is that an extra invoker thread is added. The invoker invokes the scheduler by signaling a semaphore initialized by the scheduler.

It is possible that there is no packet waiting for processing within the message queue of the processQ thread that is selected to run. This is called idle scheduling. In order to count the number of idle schedulings, the scheduler uses a specific counter for each processQ thread.

When the *Generator* process creates packets based on the Packet Group Ratio, in which the group's Packet Group Proportions are not the same as their Share, the number of idle schedulings will increase. If the Packet Group Proportion of a packet group is much lower than its group's Share, it is likely that this group's processQ thread has had more chance to run, but the thread has not had a packet to process. If there is no packet waiting for processing, the CPU share should be given to another thread that has packets waiting to be processed.

Figure 4-6 shows the complete pseudo code for the scheduler, using the Dynamic-Priority scheduling policy.

Consider an example in which the Packet Group Proportions for the three groups are 20% (Gold), 30% (Silver) and 50% (Bronze), and the Shares are: 50% (Gold), 30% (Silver) and 20% (Bronze). Because the Share of the Gold group is the highest, so processQ1 gets a greater chance of being set to the highest priority.

```
Wakeup
    Set the priority of all the scheduled processQ threads back to the lowest
    Update the CPU usage for each processQ thread
    IF the scheduled processQ thread consumes some CPU time (there were packets in the
    corresponding group waiting to be processed)
        Compare the CPU usage for each processQ thread
        Find the group with the smallest ratio of CPU usage and Share
        Choose this group's corresponding processQ thread as next scheduling thread
    ELSE (There was no waiting packet for the group)
        IF only one processQ thread did not consume CPU at previous scheduling period
            Choose the other processQ thread as next scheduling thread
        ELSE IF both the processQ threads did not consume any CPU time
            Compare the CPU usage for each thread
            Find the thread with smallest ratio of CPU usage and Share
            Choose this thread as next scheduling thread
            Reset the idle scheduling counters for each processQ thread
        ELSE (all the remaining processQ threads did consume CPU time)
            Compare the CPU usage for the other two processQ threads
            Find the thread with next smaller ratio of CPU usage and Share
            Choose this thread as next scheduling thread
        ENDIF
    ENDIF
    Raise the priority of the processQ thread that is chosen as next scheduling thread to 152
    IF this processQ thread has already consumed more CPU than its group's Share
        Decrease the scheduling time interval to half of the original amount
        Compare the CPU usages of the other two processQ threads
        Find the processQ thread with smaller ratio of CPU usage and Share
        Raise the priority of this processQ thread as well to 151
    ENDIF
Sleep for the scheduled time interval
```

Figure 4-6 Scheduling Algorithm for the Dynamic-Priority Scheduling Policy

However, the Packet Group Proportion of the Gold group is the lowest of the three groups, and processQ1 has a greater chance of being idle when it has been set at the

highest priority. Because of this idle processing within the Gold group, the scheduler gives the CPU share to either processQ2 or processQ3 to let them process the packets in their corresponding groups (Silver or Bronze).

4.5.4.4 Adaptive

The initial setting for this policy is the same as that of Fixed Priority. When the scheduler wakes up, it first updates the CPU usage for each processQ thread. The scheduler then compares the thread CPU usage to its corresponding packet group's Share. The thread with the smallest ratio of CPU usage and Share is set to the highest priority; the thread with the greatest ratio of CPU usage and Share is assigned the lowest priority; and the third processQ thread is set to the middle priority. The pseudo code for the scheduler with Adaptive scheduling policy is presented in Figure 4-7.

```
Wake up
        Update CPU usage of each processQ thread
        Compare each group's CPU usage with its Share
        Find the processQ thread with the smallest ratio of CPU usage and Share
        Raise the priority of this thread to the highest level (152)
        Find the processQ thread with the highest ratio of CPU usage and Share
        Change the priority of this thread to the lowest level (158)
        Change the priority of the third processQ thread to the middle level (155)
        Adjust schedule-interval
Sleep for the scheduled time interval
```

Figure 4-7 Scheduling Algorithm for the Adaptive Scheduling Policy

Using the same example as the one in the previous section, since the processQ1 thread has the smallest ratio of CPU usage and CPU Share, it has been set at the highest

priority. The processQ3 thread has the highest ratio of CPU usage and Share, and it has been set at the lowest priority. The processQ2 thread has been set at medium priority.

4.6 Results of Experiments

The experiments described in this section are intended to study the different factors that affect the packet drop rate and roundtrip packet processing time. The primary focus is on packet-dropping policies and scheduling policies. Given a fixed packet arrival rate, the overall drop rate is fixed in the same way that the physical link capacity is fixed. However, using different dropping policies and scheduling policies, different packet drop rates are achieved by different groups. Section 4.6.1 presents four different factors with different impacts on the packet drop rate. By assigning more CPU share to one packet group, we expect that the packets from that group will be processed faster than others with a relatively lower share of CPU resource. The factors that affect the packet roundtrip processing time are discussed in Section 4.6.2.

4.6.1 Packet Drop Rate

As the packets flow into the router, the *Node* will process each of them and send them out to the appropriate router through the physical link. The packet is processed by one of the processQ threads. With a fixed link capacity for each physical link, the incoming packet might need to be dropped if there is no available bandwidth. This section studies the effect of different factors on the drop rate of each group of packets. These factors include packet arrival rate, packet dropping policy, scheduling policy and packet processing time. For testing the differentiation of packet drop rate with different

dropping and scheduling policies, the total link capacity between any two nodes is fixed at 2MB/s.

4.6.1.1 Effect of Packet Arrival Rate

With fixed link capacities, the *Node* starts to drop packets when the arrival rate is at a certain level, one which exceeds the capacity. The input data for this test is presented in Table 4-2.

Table 4-2 Input Data for the Experiment Investigating the Effect of Packet Arrival Rate

	Gold	Silver	Bronze
Packet Group Ratio	6	6	6
Link-Share (%)	All Share (not specified for each group)		
Packet arrival rate (P/s)	varies		

As expected, the packet drop rate increases as the packet arrival rate increases. Figure 4-8 shows the relationship with a Default-Link dropping policy and the Equal-Priority scheduling policy. It will be used as a basis of comparison in the following sections. As all the packet groups have the same packet drop rate, so the overall packet drop rate shown in Figure 4-8 is the same as the packet drop rate of every single group. The packet arrival rate is usually used together with other parameters to study the network and computer performance. The following sections present a further analysis of packet drop rate.

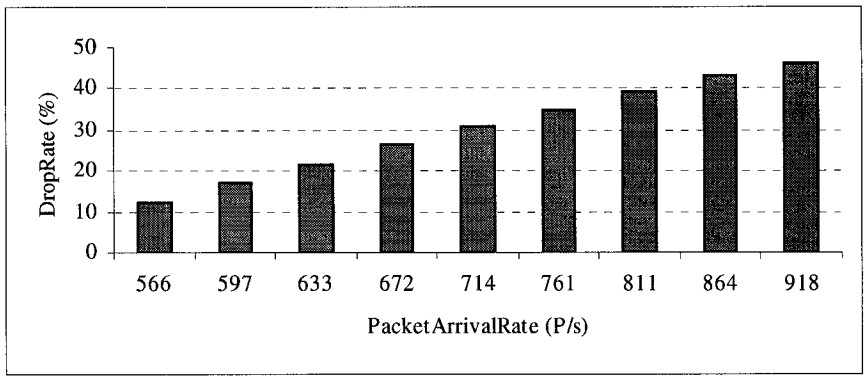


Figure 4-8 The Effect of Packet Arrival Rate on Drop Rate

4.6.1.2 Effect of Packet Dropping Policies

This section reveals the influence of different dropping policies on the drop rate.

- **Fixed-Link Dropping Policy**

  With the Fixed-Link dropping policy, the packet drop rate of each group depends on its occupied portion of the full link capacity and its Packet Group Ratio. The input data for this test is shown in Table 4-3.

Table 4-3 Input Data for Experiment with Fixed-Link Dropping Policy

	Gold	Silver	Bronze
Packet Group Ratio	6	6	6
Link-Share (%)	50	30	20
Packet arrival rate (P/s)	varies		

When the Packet Group Ratio for each group is fixed, the drop rate of one group decreases when its allocated link capacity portion increases, as there will be more bandwidth available to be used to process the arriving packets. Figure 4-9 shows the different packet drop rates for each group.

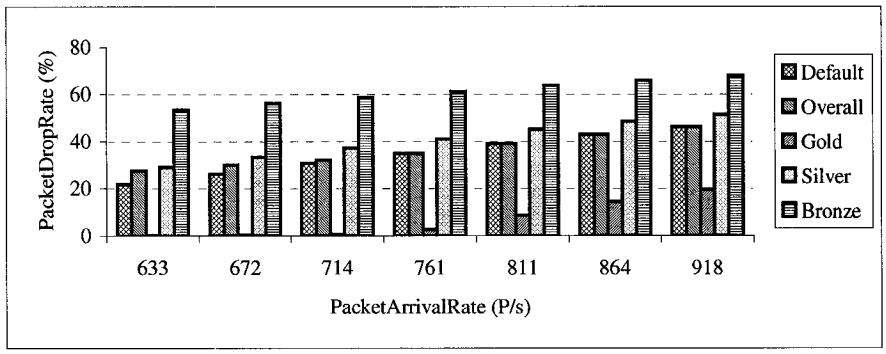


Figure 4-9 Default-Link vs Fixed-Link

In Figure 4-9, 'Default' represents the drop rate with the Default-Link dropping policy, and 'Overall' represents the overall drop rate for the Fixed-Link dropping policy. Gold, Silver and Bronze represent the packet drop rates for each single group. The Default is also equivalent to the Fixed-Link when the Link Capacity is divided equally into the three groups. When the Packet Arrival Rate exceeds the link capacity of the node, the packets start being dropped. Since the group arrival rate is the same for all groups, the drop rate of each group is only related to its Link-Share. For example, the Gold packet group has the highest Link-Share; thus the rate of packets sent out from this packet group will be the highest and the drop rate the lowest. The Silver packet group is allocated 30% of link capacity. This is close to, but a little less than, its Packet Group Proportion (33.3%), so its drop rate is a little bit higher than the Default-Link drop rate. Because the Bronze group only has 20% of the link capacity, which is much less than its Packet Group Proportion, the drop rate is much higher than the Default. In the real world, the most important customers should get a higher share of the

link capacity than ordinary customers, when they need it. However, the main problem with this approach is that the link capacity may be under-utilized if the highest priority group does not use up all its Link-Share. As shown in Figure 4-9, when the arrival rate is less than 714 packets per second, the lower priority packet groups (Bronze and Silver) cannot use the available bandwidth that is reserved for Gold, even if the Gold group traffic is lower than its capacity.

When the packet arrival rate for each group is fixed, if we increase the link capacity share for one group, its drop rate decreases. Table 4-4 shows the input data for testing the effect of various Link-Share distributions.

Table 4-4 Input Data for Experiment with Various Link-Shares (Fixed-Link)

	Gold	Silver	Bronze
Packet Group Ratio	6	6	6
Link-Share (%)	varies	varies	varies
Packet arrival rate (P/s)	811		

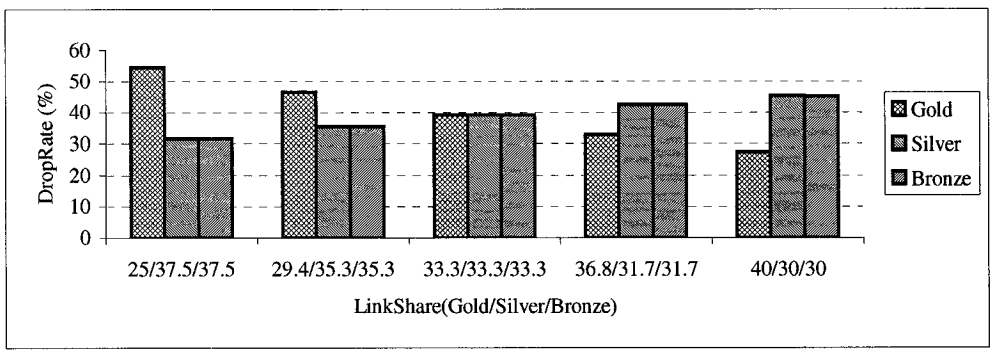


Figure 4-10 The Effect of Link-Shares on Drop Rate

As expected, Figure 4-10 shows the drop rate for Gold packets is decreased when the Link-Share of Gold group is increased from 25% to 40%,

while the drop rates of both Silver and Bronze packets are increased as their Link-Shares are reduced from 37.5% to 30%.

Similarly, if the Link-Share for each packet group is fixed, increasing the Packet Group Proportion of one group results in a higher group drop rate, because the same available bandwidth will be shared by the packets that continue to arrive. Table 4-5 shows the input data for the test on investigating the impact of different Packet Group Ratios.

Table 4-5 Input Data for Experiment with Various Packet Group Ratios (Fixed-Link)

	Gold	Silver	Bronze
Packet Group Ratio	varies	varies	varies
Link-Share (%)	50	30	20
Packet arrival rate (P/s)	811		

The results for the sets of different Packet Group Ratios are shown in Figure 4-11.

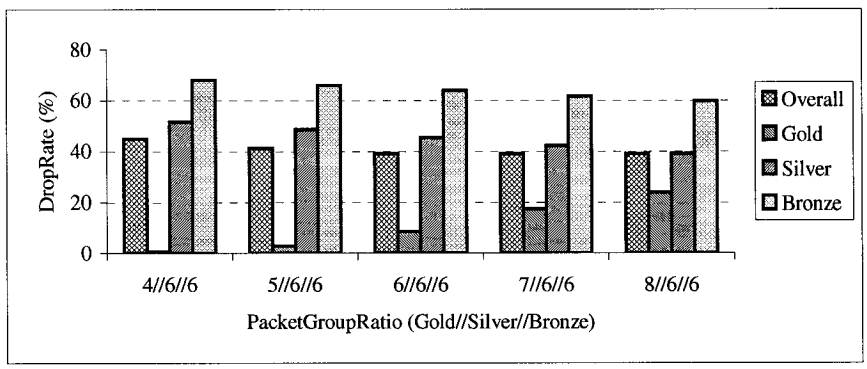


Figure 4-11 The Effect of Packet Group Ratios on Drop Rate

In Figure 4-11, because the rate of packets sent out in the Gold group is fixed, its drop rate decreases as its Packet Group Proportion becomes smaller.

When the Packet Group Proportion of the Gold group decreases from 40% to 25%, the Packet Group Proportions for both Silver and Bronze groups increase from 30% to 37.5%, so their drop rates increase slightly.

- **Dynamic-Link**

Dynamic-Link dropping policy tries to avoid the situation of under-utilized capacity that was discussed in the previous section. The Fixed-Link dropping policy may waste some of the bandwidth designated for the Gold packet group when its group packet arrival rate is such that its assigned portion of link capacity cannot be utilized. Section 4.3.3 illustrates how Dynamic-Link policy works. The input data for the Dynamic-Link dropping policy test is presented in Table 4-6. In Table 4-6, n1 means that n1 percent of link capacity is reserved for the Gold group; n2 means that n2 percent of link capacity is reserved for both the Gold and Silver packet groups.

Table 4-6 Input Data for Experiment with Dynamic-Link Dropping Policy

	Gold	Silver	Bronze
Packet Group Ratio	6	6	6
Link-Share (%)	up to 100	up to 100-n1	up to 100-n1-n2
Packet arrival rate (P/s)	633		

The detailed testing results are shown in Figure 4-12. In this figure, the horizontal axis shows how the link capacity is distributed. Fixed-Link shows the group packet drop rates with the Fixed-Link dropping policy (See Figure 4-9). Default shows the three groups of packets equally sharing the full link capacity

with Default-Link dropping policy. The n1+n2 pairs represents the percentage threshold reserved for the Gold and Silver groups (see Table 4-6). Of these pairs, n1 percent of the link capacity is reserved for the Gold group, and the other n2 percent of link capacity is reserved for both the Gold and Silver groups, while the rest of the link capacity (100-n1-n2) is shared by all three groups.

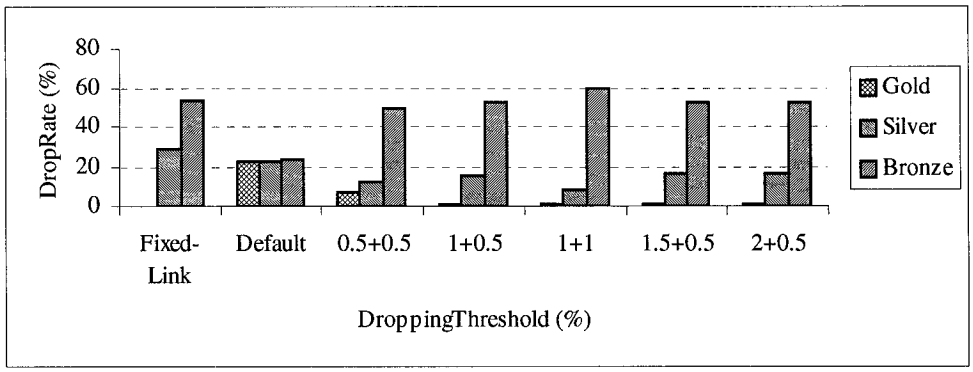


Figure 4-12 The Effect of Dropping Threshold on Drop Rate

Figure 4-12 shows that the Dynamic-Link is comparable to the Fixed-Link when the full link capacity is split into three portions, as follows: 2% is reserved for Gold group packets only, 0.5% is reserved for packets in both Gold and Silver groups, and 97.5% is shared by the packets in all the groups. Compared with the results of Fixed-Link, the Dynamic-Link dropping policy achieves a lower drop rate than the Fixed-Link dropping policy for both the Silver and Bronze groups of packets, while the Gold group is close to no dropping.

4.6.1.3 Effect of Scheduling Policies

Section 4.4 describes various scheduling policies. The study presented in this section shows that the variation of scheduling policies has less influence on the overall

drop rate when integrated with the Default-Link and Dynamic-Link dropping policies. This is because all the packet groups share the overall bandwidth. However, with a Fixed-Link dropping policy, the drop rate varies when the differences in Packet Group Proportions between the groups become large.

The packet drop rate achieved with different scheduling and packet dropping policies is discussed next.

- **Scheduling policies with Default-Link dropping policy**

  Although the overall packet drop rates are close to each other, the drop rate of every single group of packets varies when different scheduling policies are used. With lower arrival rates, the differences among various scheduling policies are negligible, so we adopt a higher arrival rate to compare these scheduling policies. Higher arrival rates could occur when there is a sudden increase in traffic due to the increase in the number of users, or network failures and traffic reroute. The input test data is shown in Table 4-7.

Table 4-7 Input Data for Experiment Investigating the Effect of Scheduling Policies

	Gold	Silver	Bronze
Packet Group Proportion (%)	40	30	30
Share (%)	40	30	30
Link-Share (%)	All Share		
Packet arrival rate (P/s)	918		

Figure 4-13 shows the group drop rates using different scheduling policies and the Default-Link dropping policy. The Fixed-Priority scheduling policy starves the lowest priority packets that lead to the highest drop rate for Bronze group. The Equal-Priority scheduling policy serves all groups of packets with a

similar drop rate for each group. Both the Dynamic-Priority and Adaptive scheduling policies have a drop rate close to that of Equal-Priority, because their group's Share is the same as their Packet Group Proportion.

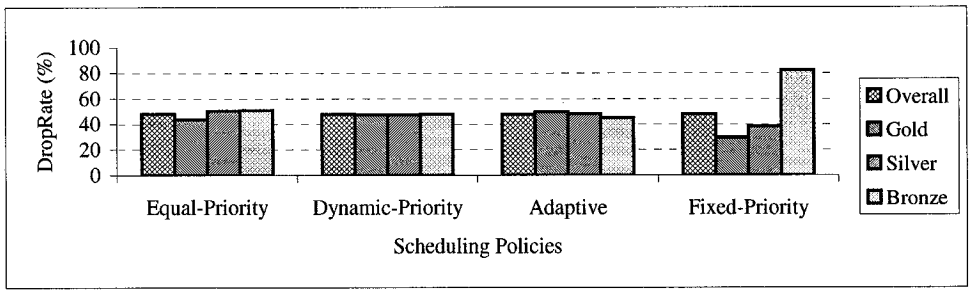


Figure 4-13 The Effect of Scheduling Policy on Drop Rates with the Default-Link Packet Dropping Policy

Figure 4-14 shows the changes in drop rate when the Base Scheduling Interval (described in Section 3.2.2) changes with the Adaptive scheduling policy.

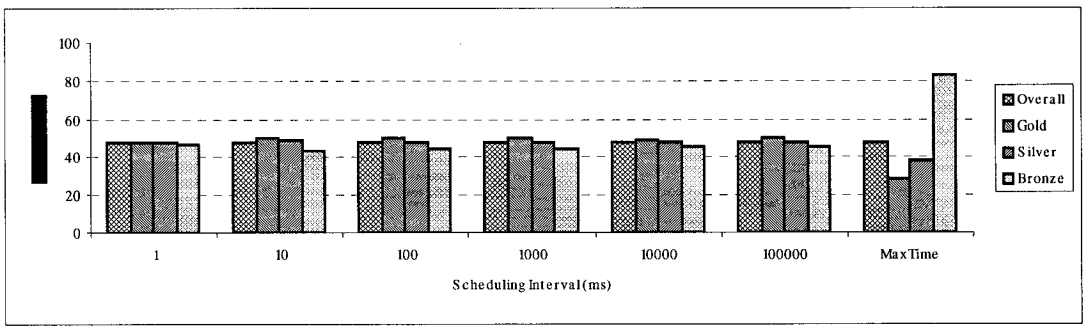


Figure 4-14 The Impact of Base Scheduling Interval for Adaptive Scheduling Policy on Drop Rate

As we can see, when the scheduling interval increases, the CPU time used by the scheduler per second decreases, and the difference in drop rate between the packet groups therefore becomes higher. When the scheduling interval is larger than or equal to the total run time for the experiment (Max Time), there is essentially no scheduling. It is then equivalent to the Fixed Priority scheduling, which is also illustrated in Figure 4-13.

- **Scheduling policies with Fixed-Link dropping policy**

When the packet arrival rate is low compared to the rate that achieves the full link capacity of the physical link, the effect of the scheduling policy is not obvious. Table 4-8 provides a summary of the input data for a test results of which are presented in Figure 4-15.

Table 4-8 Input Data for Experiment in which Link-Share is Equal to Packet Group Proportion

	Gold	Silver	Bronze
Packet Group Proportion (%)	varies		
Share (%)	Same as Packet Group Proportion		
Link-Share (%)	Same as Share		
Packet arrival rate (P/s)	633		

Figure 4-15 shows the overall drop rate for each scheduling policy when the Share is the same as the Packet Group Proportion. Considering these packets are evenly distributed between two routers, the packets go to each link at a rate of 2.53 (633 packets * 1000 bytes/packet * 8 bits/byte / 2 routers) MB/s.

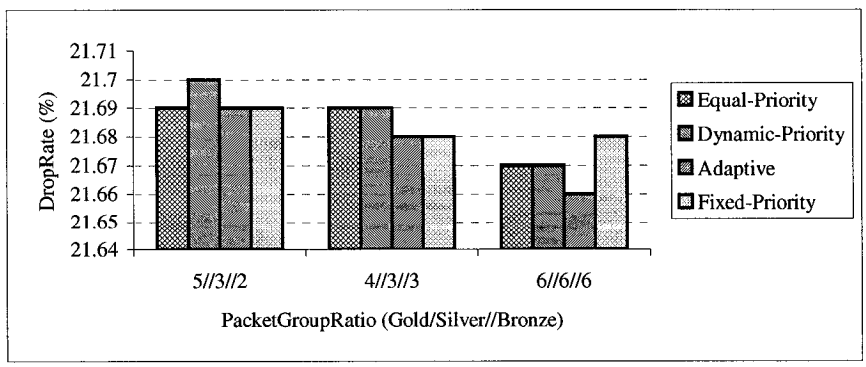


Figure 4-15 Drop Rate with Proportional Shares

Figure 4-16 shows the drop rates when the group's Share is not the same as its Packet Group Proportion (Packet Group Proportion equals the Link-Share). As shown in Figure 4-16, the Share for the Gold group increases from 25% on the left to 50% on right hand side. Both Figure 4-15 and 4-16 show that the overall drop rates are close to each other for all scheduling policies. The actual differences are less than 0.2%.

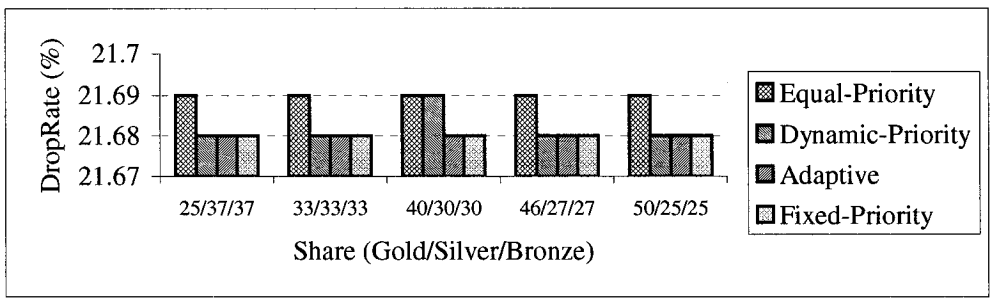


Figure 4-16 Drop Rate with None-Proportional Shares

However, when the packet arrival rate increases to a higher level, the Fixed Priority scheduling policy can cause bandwidth to be wasted. Because of starvation of the processQ thread that processes the Bronze packets, the bandwidth assigned to the Bronze group may not be fully utilized. The input data for this testing is shown in Table 4-9.

Table 4-9 Input Data for Experiment Investigating Starvation

	Gold	Silver	Bronze
Packet Group Ratio	varies	varies	varies
Link-Share (%)	Same as Packet Group Proportion		
Packet arrival rate (P/s)	811		

Figure 4-17 shows the effect of Packet Group Ratio on the drop rate. The traffic rate on each link is 3.24 MB/s, and when the Packet Group Proportion for the Bronze group drops to about 5%, the drop rate of the Bronze group increases significantly due to starvation of the processQ thread that processes this packet group.

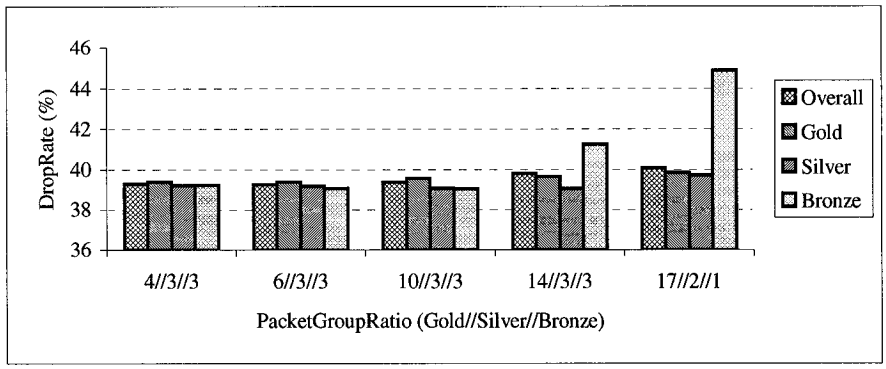


Figure 4-17 The Effect of Packet Group Ratio on the Drop Rate for Fixed-Priority Scheduling

4.6.1.4 Effect of Packet Processing Time

The section presents the effect of packet processing time on the packet drop rates. The packet processing time is the time used for the processQ thread to process a packet. The study is useful for better understanding the influence of computer processing speed on network performance. The input data for this study is presented on Table 4-10.

Table 4-10 Input Data for Experiment with Various Packet Processing Times

	Gold	Silver	Bronze
Packet Group Proportion (%)	40	30	30
Share (%)	40	30	30
Link-Share (%)	40	30	30
Packet arrival rate (P/s)	918		

Table 4-11 shows the difference in dropping at different locations with different processing times (shown as Ptime in the Table 4-11). The overall drop rates are close to each other, ranging from 47.56% to 48.04%. In the 'Drop Location' column in Table 4-11, PreDrop means the packet was dropped by the processQ thread because the queuing time exceeded the Maximum Packet Waiting Time, while SendOut means the packets were dropped before being sent out from the destination thread due to lack of bandwidth. The entries in the table for a given scheduling policy are the percentages of drop-outs of the overall dropped packets at each location.

Table 4-11 The Effect of Packet Processing Time on Drop Rates Achieved at Different Locations

Ptime(μs)	Drop Location	Equal-Priority	Dynamic-Priority	Adaptive	Fixed-Priority
300	PreDrop	46%	59%	35%	41%
	SendOut	54%	41%	65%	59%
500	PreDrop	45%	64%	41%	44%
	SendOut	55%	36%	59%	56%
700	PreDrop	47%	69%	45%	50%
	SendOut	53%	31%	55%	50%

For example, out of all the dropped packets for Equal-Priority when packet processing time equals 300 μs, 46% of these packets were dropped from the processQ threads. The other 54% were dropped from the destination threads.

As the packet processing time is increased, the time each packet stays in the router becomes longer, and the packet queuing time increases as well. This leads to more packets being dropped by the processQ threads because they exceed the Maximum

Packet Waiting Time. For example, for Dynamic-Priority scheduling policy, as the processing time increases from 300 μs to 700 μs, the proportion of packets dropped by the processQ thread out of the total dropped packets increases from 59% to 69%. This means that more packets were dropped because the buffer is full.

4.6.2 Mean Processing Time and Mean Roundtrip Time

A packet travels along a sequence of routers during its lifetime. One metric that is used for performance evaluation is mean roundtrip time. To obtain a more accurate roundtrip time from the experiment, we minimize the impact of network delay by giving preference to the packets arriving at the destination, so that they can be sent back to the original source right away. Based on this scheme, all the results from the experiments conducted in this study show that the mean roundtrip time is only slightly higher than the mean processing time. Therefore, the influence of scheduling policies can be measured using either mean processing time or mean roundtrip time, because they are close to each other. The Mean Roundtrip Time is reported in the figures that follow.

Of all the different schedulers, the Adaptive scheduler, as expected, gives the shortest Mean Roundtrip Time. Due to frequent scheduling, the Dynamic-Priority scheduling policy results in the longest Mean Roundtrip Time.

Both the Equal-Priority and Fixed-Priority scheduling policies do not count the Share at all. The dominating factor of the Equal-Priority scheduling policy is Packet Group Ratio, while the main concern of the Fixed-Priority scheduling policy is giving the highest priority to the top-level (Gold) packets.

The following two sections present the testing results to show how the mean round trip time is affected by a scheduling policy when CPU share is assigned to each group. Section 4.6.2.1 shows the effect of different scheduling policies on performance when the Share of each group is the same as its Packet Group Proportion. Section 4.6.2.2 shows the effect of different scheduling policies on performance when the Shares of the groups are different from their Packet Group Proportions.

4.6.2.1 Share is the Same as the Packet Group Proportion

Each scheduling policy has its own means of distributing the CPU resource among the three packet groups. However, when the Shares of the three groups are the same as their Packet Group Proportions, except for the Fixed Priority scheduling policy, the Mean Roundtrip Time of each packet group is very similar to the others. The input data for the test that captures the effect is shown in Table 4-12.

Table 4-12 Input Data for Experiment in which Link-Share is Equal to Share

	Gold	Silver	Bronze
Packet Group Proportion (%)	40	30	30
Link-Share (%)	40	30	30
Share (%)	40	30	30
Packet arrival rate (P/s)	633		

Figure 4-18 shows the detailed results. The Share of each group is exactly the same as the group's Packet Group Proportion. As illustrated in Figure 4.18, except for the Fixed-Priority scheduling policy, with the same Share the roundtrip times for a given scheduling policy are about the same across packet groups. For the Fixed Priority

scheduling policy, the Gold group has the shortest roundtrip time, as those packets are given the highest priority when processed.

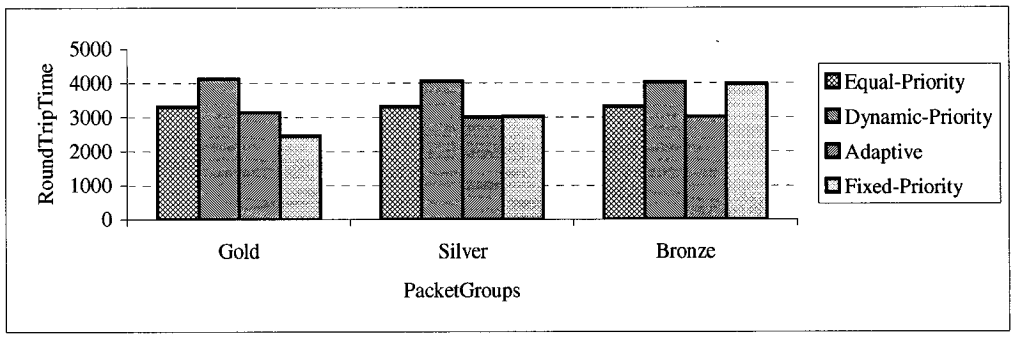


Figure 4-18 Round Trip Time with Link-Share is Equal to Share

Running the scheduler frequently results in the highest packet roundtrip time with the Dynamic-Priority scheduling policy. In other words, the Dynamic-Priority scheduling policy has higher computational overhead.

4.6.2.2 Share Inequal to Packet Group Proportion

When the CPU Share of a group is not the same as its Packet Group Proportion, the scheduling policies perform very differently. The input data for this test is presented in Table 4-13.

Table 4-13 Input Data for Experiment in which Link-Share Inequal to Share

	Gold	Silver	Bronze
Packet Group Proportion (%)	40	30	30
Link Distribution (%)	40	30	30
Share (%)	varies	varies	varies
Packet arrival rate (P/s)	633		

Figure 4-19 shows the variation of the roundtrip time of the Gold group when its Share varies. With an Equal-Priority scheduling policy, all three groups of packets have a similar roundtrip time. With a Fixed-Priority scheduling policy, the Gold group has the shortest roundtrip time, while the Bronze group has the longest.

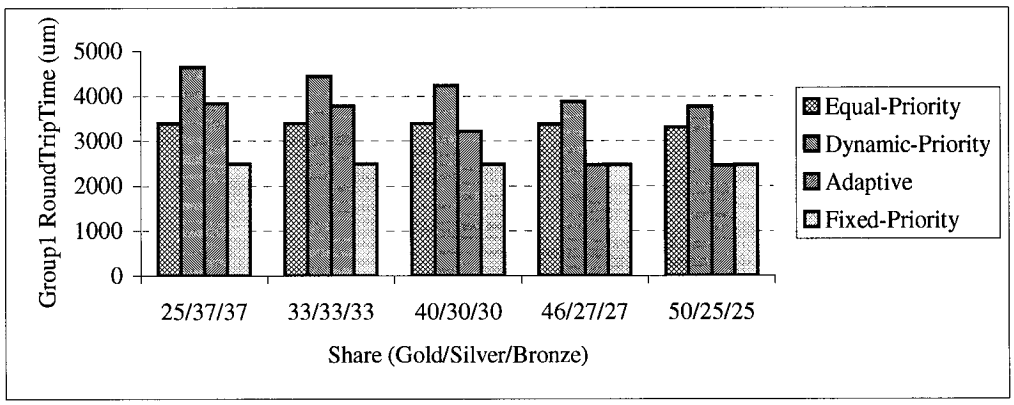


Figure 4-19 Round Trip Time with Link-Share Inequal to Share

Both the Dynamic-Priority and Adaptive scheduling policies distribute the CPU resource based on the Share of each packet group. If a packet group has a higher Share compared to its Packet Group Proportion, then most of its packets have more chance of being processed at a higher priority, thus shortening the roundtrip time. In contrast, if a packet group has a lower Share than its Packet Group Proportion, its packets will most likely be processed at a lower priority, so the roundtrip time will be longer.

If the Share of one packet group is much lower than its Packet Group Proportion, the packets in that group will have less chance of being processed. The input data for the test that investigates this in the context of the Adaptive scheduler is presented in Table 4-14.

The results in Figure 4-20 show the variation of the roundtrip time of each group when the Share is changing.

Table 4-14 Input Data for Experiment with the Adaptive Scheduler

	Gold	Silver	Bronze
Packet Group Proportion (%)	40	30	30
Link-Share (%)	Equals to the Share		
Share (%)	varies	varies	varies
Packet arrival rate (P/s)	811		

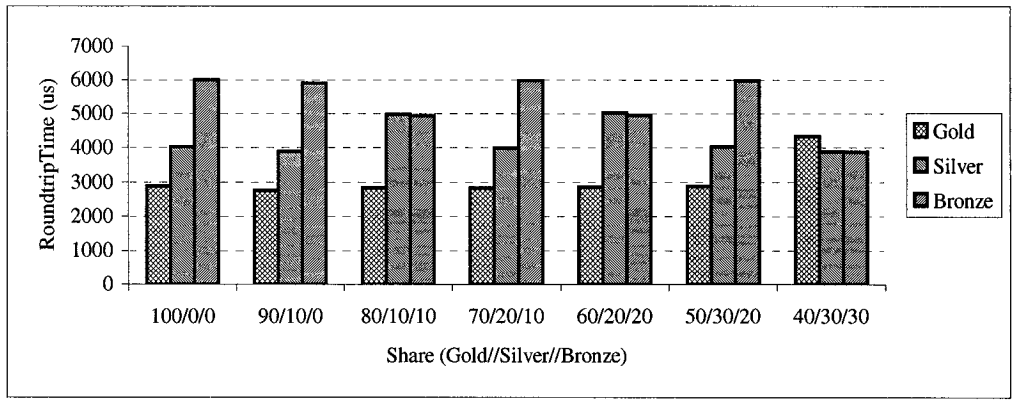


Figure 4-20 Group Roundtrip Times

4.7 Summary

In this chapter, we investigated the effects of different dropping and scheduling policies on the performance prototype of network routers.

As the dropping policy varies, the drop rate of each individual packet group changes to a certain degree. The characterizations of the different dropping policies are summarized as follows:

- Default-Link does not provide any differentiated service for clients. The router starts to drop packets when the outgoing packets exceed the link capacity

between the connected routers. As the network link is shared by all the groups of packets, the increasing packet arrival rate for each packet group not only affects its own group drop rate but also the drop rates of the other two groups.

- Fixed-Link can provide different classes of service by precisely splitting the network link capacity for the different packet groups. Because the bandwidth allocated to each group of packets is fixed, the drop rate of one packet group is strictly related to its own packet arrival rate. To increase the level of service for one group of packets, we can simply assign more bandwidth to it. However, assigning too much bandwidth to a group may result in bandwidth wastage.
- Dynamic-Link provides the flexibility to split the network resource more accurately to fit each group's needs. Because only a small portion of the resource is reserved for the higher priority groups, bandwidth will not be wasted in most cases. Adjusting the reserved portion can give the requested level of service to the group with higher priority (See Figure 4-12).

Different scheduling policies can affect the overall router performance. Distributing the CPU resource to three groups in different ways will result in different routing delays for each group of packets. These differences are summarized below:

- The Equal-Priority scheduling policy does not affect the router performance. All packet groups are treated the same, so they have the same roundtrip time.
- The Fixed Priority scheduling policy gives rise to the shortest delays for the highest priority group (Gold) of packets. However, this could cause the starvation of the lowest priority packet group (Bronze) and increase its group drop rate.

- Both Dynamic-Priority and Adaptive scheduling policies can give each packet group a reasonable delay by adjusting the priorities of different threads that handle each packet group. But the Adaptive scheduling policy produces the shortest overall delay, as it uses less time to perform scheduling.

Overall, in practice, different classes of service for packet groups can be effectively achieved by a combination of the Dynamic-Link dropping policy and the Adaptive scheduling policy. The Fixed-Link dropping policy could also be used because of its simplicity, but bandwidth wastage should be avoided by regularly changing the portions of the network link capacity reserved for different packet groups. The Fixed-Priority scheduling policy can be used only if the least important packet group can be ignored at any time.

Chapter 5 Conclusions

This thesis consists of two main parts. The first part studied the Fair-Share Scheduling of the CPU resource, while the second part presented a discussion of scheduling policies and packet dropping policies in a router. A ChorusOS based performance prototype was used in both investigations. A characterization of the scheduling behavior of ChorusOS was also performed.

5.1 Summary

In order to establish the fair sharing of the CPU resource, four schedulers were evaluated in this research: Equal-Priority, Static, Dynamic-Priority and Adaptive. As the workload varies, the schedulers distribute the CPU resource differently. The highlights of each scheduling policy are presented.

- The Equal-Priority Scheduler gives the user total control over the CPU usage. It can only achieve fair share when the maximum utilization of each group is exactly the same as its Share.
- The Static Scheduler picks up one group of jobs during each scheduling interval. It can only achieve fair sharing when the maximum utilization of each group is larger than its Share.

- The Dynamic-Priority Scheduler schedules each group of jobs dynamically. It achieves a fair sharing of the CPU resource with a relatively high overhead, and thus causes a drop in overall system utilization.
- The Adaptive Scheduler schedules each group of jobs in every scheduling period. It provides Fair Sharing of the CPU resource with minimal scheduling overhead.

To apply the schedulers to the router investigation, we examined four different scheduling policies: Equal-Priority, Dynamic-Priority, Adaptive and Fixed-Priority. The first three have already been described. For the Fixed-Priority scheduling, the packet with the highest priority will experience the shortest possible delay, while the lowest priority packet will have the longest delay, or may even be starved. Note that the workload used in Chapter 3 consists of jobs whereas that for the network router described in Chapter 4 it consists of data packets.

At the same time, we also examined three packet dropping (bandwidth distribution) policies in the performance prototype of a network router. They were: Default-Link, Static-Link and Dynamic-Link.

- The Default-Link dropping policy follows the First-In-First-Out sequence for all the incoming packets, with no consideration given for differentiated service to packet groups at all.
- The Fixed-Link dropping policy can provide differentiated service by precisely splitting the network link capacity into three portions, one for each packet group. However, over-assigning bandwidth to a group will result in

bandwidth wastage, because the extra bandwidth cannot be used by other groups.

- Dynamic-Link dropping policy provides the flexibility to split the network resource more accurately to fit each group's needs. Precisely adjusting the reserved portion based on demand can provide the requested level of service for packets with higher priority.

5.2 Conclusions and Future Work

Given a number of jobs to execute in a system, the CPU resource is split into the same number of portions as the number of jobs. Compared to the Equal-Priority scheduler, which employs the FCFS scheduling policy of the ChorusOS, all three schedulers (Static, Dynamic-Priority and Adaptive) achieve a higher degree of fairness. Of these three, the Adaptive scheduler provides the best fair share by using the following simple techniques:

- The next scheduling is undertaken within an adjusted time quantum. This counter-balances the extra overhead caused by the Dynamic-Priority scheduler.
- All the groups of jobs are scheduled during each scheduling period. The priority of a job group depends on the ratio of its current GCU and Share. Smaller the ratio, higher the priority. Thus a job group whose GCU is much lesser than its Share has more opportunity to run.

When investigating a network router, the issue of how to distribute the link capacity becomes very important. The Default-Link strategy does not provide any differentiated service. The drop rate of Gold for the Dynamic-Link is close to zero

whereas the drop rates for Silver and Bronze are lower than those achieved with the Fixed-Link strategy (See Figure 4-12).

With a fixed arrival rate, different scheduling policies result in different delays for each packet group. Similar to the Adaptive Scheduler in the Fair-Share Scheduling study, the Adaptive scheduling policy gives rise to the shortest delay for the packet group with the highest Share.

The experiments described in this thesis have concerned a workload with 3 groups of jobs. The scheduling techniques and the packet dropping policies, however, can be extended to handle a larger number of job groups. These techniques could be implemented using operating systems that have preemptive priority scheduling policies which are similar to ChorusOS. The study could be useful in real world applications that require differentiated service. For instance, audio/video packets should be given a higher share of CPU to decrease latency, while data packets can be given a lower CPU share.

Future work should address the areas of automatic process scheduling and scalability.

- These policies should be tested in the context of real applications.
- The network traffic varies from time to time. It is possible that one scheduling policy will perform well with certain workloads, while another will perform well in other situations. A hybrid process scheduler will allow the system to switch among all these potential scheduling policies in order to achieve the best overall performance. Such a hybrid scheduler needs further investigation.

References

1. J.C.R. Bennett and H. Zhang, "Hierarchical Packet Fair Queuing Algorithms", *IEEE/ACM Transactions on Networking*, Vol. 5, No. 5, October 1997, pp. 675 – 689.

2. S. Bodamer, "A New Scheduling Mechanism to Provide Relative Differentiation for Real-Time IP Traffic", In *Proceedings of the IEEE Global Telecommunications Conference (GlobeCom '00)*, San Francisco, November 2000, pp. 646 – 650.

3. Cisco Systems, "Class-Based Weighted Fair Queuing", Jan 2003, http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120t/120t5/cbwfq.pdf

4. Cisco Systems, "Planning for Quality of Service", http://www.cisco.com/en/US/products/sw/cscowork/ps2064/products_user_guide_chapter09186a00800807f7.html

5. Cisco Systems, "RFC 1812: Requirements for IP Version 4 Routers", June 1995, http://www.faqs.org/rfcs/rfc1812.html

6. J.A. Cobb, M.G. Gouda, and A. El-Nahas, "Time-Shift Scheduling – Fair Scheduling of Flows in High-Speed Networks", *IEEE/ACM Transactions on Networking*, Vol. 6, No. 3, June 1998, pp. 274 – 285.

7. D.H.J. Epema and J.F.C.M. de Jongh, "Proportional-Share Scheduling in Single-Server and Multiple-Server Computing Systems", *Performance Evaluation Review*, Vol. 27, Issue 3, December 1999, pp. 7 – 10.

8. S. Floyd and V. Jacobsen, "Link-Sharing and Resource Management Models for Packet Networks", *IEEE/ACM Transactions on Networking*, Vol. 3, No. 4, August 1995, pp.365 – 386.

9. B. Ford and S. Susarla, "CPU Inheritance Scheduling", In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA (USA), October 1996, pp. 91 – 106.

10. M. Gien, "From Operating Systems to Cooperative Operating Environments", *Chorus Systems*, October 1992, http://www.jaluna.com/developer/papers/CS-TR-92-82.pdf.

11. P. Goyal, X. Guo, and H. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems", In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle WA (USA), October 1996, pp. 107 - 121.

12. P. Goyal, H. M. Vin, and H. Cheng, "Start-Time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks", *IEEE/ACM Transactions on Networking*, Vol. 5, No. 5, October 1997, pp. 690 – 704.

13. K. Jeffay, F.D. Smith, A. Moorthy, and J. Anderson, "Proportional Share Scheduling of Operating System Services for Real-Time Applications", In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998, pp. 480 – 491.

14. J. Kay and P. Lauder, "A Fair Share Scheduler", *Communications of the ACM*, Vol. 31, No. 1, January 1988, pp. 44 – 55.

15. K.K. Leung, "An Execution/Sleep Scheduling Policy for Serving an Additional Job in Priority Queuing Systems", *Journal of the Association for Computing Machinery*, Vol. 40, No. 2, April 1993, pp. 394 – 417.

16. Nortel Networks, "CGNet: A users guide & designers manual", June 2001.

17. L.L. Peterson and B.S. Davie, *Computer Networks*, Morgan Kaufmann Publishers, 1996.

18. X. Qie, A. Bavier, L. Peterson, and S. Karlin, "Scheduling Computations on a Software-Based Router", In *Proceedings of the ACM SIGMETRICS '2001 Conference*, Cambridge, Massachusetts, USA, June 2001, pp. 13 – 24.

19. J. Regehr, "Some Guidelines for Proportional Share CPU Scheduling in General-Purpose Operating Systems", In *Work in progress session of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001),* London, UK, December 2001, pp. 53 – 56.

20. M. Shreedhar and G. Varghese, "Efficient Fair Queuing Using Deficit Round Robin", *IEEE/ACM ACM Transactions on Networking*, Vol. 4, No. 3, June 1996, pp. 375 – 385.

21. A. Snavely, D. M. Tullsen, and G. Voelker, "Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor", *Performance Evaluation Review*, Vol. 30, Issue 1, June 2002, pp. 66 – 76.

22. I. Stoica, S. Shenker, and H. Zhang, "Core-Stateless Fair Queuing: Achieving Approximately Fair Bandwidth Allocations in High Speed Networks", *Computer Communication Review*, Vol. 28, Issue 4, October 1998, pp. 118 – 130.

23. Sun Microsystems, "ChorusOS 4.0 Introduction", December 1999, http://docs.sun.com/db/doc/806-0610.

24. Sun Microsystems, "ChorusOS 4.0 Reference Manual", December 1999, http://docs.sun.com/db/coll/571.1.

25. Sun Microsystems, "ChorusOS 5.0 Application Developer's Guide", December 2001, http://docs.sun.com/db/doc/806-6894.

26. C.A. Waldspurger and W.E. Weihl, "Lottery Scheduling: Flexible Proportional-Share Resource Management", In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation.* Monterey, CA, USA, November 1994, pp. 1-11.

27. X. Yuan, "Characterization of Uniprocessor and Multiprocessor HP-UX Scheduler Behavior", Technical Report, Nortel Networks, January 1999.