Telecommunications Call Processing on a Linear Processor Array

David Knox

A THESIS

submitted to the School of Graduate Studies and Research

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF APPLIED SCIENCE

in

Electrical Engineering

Ottawa-Carleton Institute of Electrical Engineering

Department of Electrical and Computer Engineering

Faculty of Engineering

University of Ottawa

OTTAWA, ONTARIO K1N 6N5

ISBN 0-612-19976-2

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Acknowledgements

I would like to thank my supervisor, Dr. Sethuraman Panchanathan, for his advice and both Richard Cowper and Jules Meunier for their financial support. I would also like to thank Brian Baker, Terry Newell, Zaihua Luan, Dale Stribeau and J.W.J. (Bill) Williams for information and technical discussions. Finally, I would like to thank my family for their patience and encouragement throughout this work.

Abstract

New telephone features and their increasingly complex interactions with one another require better performance from Telecommunication Switching systems. Since call processing is the most computationally expensive task in a switch, multi-processor systems are being considered for the call processing cores of many commercial switch architectures. In this thesis, the design criteria for such a call processing system have been generated and analyzed. A new approach, the heterogeneous linear pipelined array is proposed and compared with a common load sharing architecture.

The linear pipelined array has been simulated using a detailed Verilog Hardware Description Language model, to further verify its suitability for call processing applications. Several fixed assignment schemes and three simple heuristic assignment schemes have been simulated and the resulting throughput and processor utilizations has been studied. A semi-static approach has been tested, where an optimal schedule is determined based on the relative quantities of calls in specific states. The performance of the array for the call processing task is shown, for a variety of cases, to be dependent on the degree of load balancing which can be achieved in the array and the extent to which processing bottlenecks can be avoided. The results for the semi-static schedules are significantly better than for the fixed schemes.

Performance is shown to be limited if there are significant delays between contiguous sequences of sub-tasks. The mechanism used to reject load is important and has a fundamental limiting effect on system throughput. Different offered loads have been simulated and the linear pipelined array performance is stable under overload conditions. The results of the analysis indicate that a heterogenous linear array can be used effectively for the call processing application and also exhibits good scaleability. The detailed simulations used here are an effective approach for the analysis of the underlying causes for events and provide a representative architectural model of the system, which can be adapted and refined as further experiments are performed.

Contents

List of Illustrations

List of Tables

Terms and Abbreviations

Active-Standby:	Architecture of two processors where one processor is active and the second is kept in reserve ready to take over should the active processor fail
Basic Call Model:	BellCore-defined representation of the stages involved in processing a call
BellCore:	Single technical body responsible for providing a single view of Telco requirements and problems
Bottleneck:	A point of congestion in a system (for a linear array, this is the processing stage which has the maximum delay)
Call Processing:	The co-ordination of resources and decision-making required to ensure that a connection between two or more distinct users can be originated, take place and then terminate properly
Circuit-Switched:	Connections made based on a nailed-up end-to-end connection
Core:	Central Processing subsystem in a telephone switch (whose main function is call processing)
CPU:	Central Processor Unit; basic processing circuitry within a processor; contains arithmetic logic unit and electronics to execute program instructions
DS1:	Standard format developed for time-division multiplexed transmission of voice data (contains 24 multiplexed voice channels and signalling information
DTMF:	Dual-Tone Multi-frequency; signalling tone format used in Telephone sets for the dialling and the recognition of user digits (replacing the older pulse dial signalling scheme)
Function Sharing System:	Architecture which has a fixed assignment of tasks or jobs assigned to specific processors, but which does not require all processors to perform identical tasks
Jobs:	A particular task which has to be completed and which is made up of a set of program modules, all of which need to be executed prior to job completion; for the purposes of this thesis, a job is equal to the total sum of processing required to handle a call
Heterogenous System:	Architecture in which processors can be different
Heuristic:	A decision rule
Homogenous System:	Architecture in which all processors are identical
ISDN:	Integrated Service Digital Network: Higher bandwidth telephone service, with separate voice and data channels

Linear Array:	Architecture which arranges processors sequentially, with connections only between neighbouring processors
Load-Sharing System:	Architecture (usually homogeneous) in which less loaded processors can be assigned work which busier processors are not able to handle
Loosely-Coupled System:	Architecture with a low degree of interaction between processors; high processor independence is permitted
LSSGR:	LATA Switching System Generic Requirements
Module:	Segment of executable instructions which can be identified or partitioned in a program (corresponds here to the series of program operations in a sub-call)
NP-complete	NP is the class of decision problems which can be solved in polynomial time by a nondeterministic machine. If a problem, R, is NP-complete, it can be shown that R is in the NP class and also that all other existing problems in the NP class are polynomial-transformable to R.
NP-hard:	If a problem, R, is NP-hard, it is only possible to show that all other existing problems in the NP class are polynomial transformable to R, but not that R is in the NP class
Partition:	A sequential set of program modules
Passthrough:	For the linear array, this means that a call is transmitted through an stage in the array without processing
PCS:	Personal Communications Services; network standards which promise to provide a higher bandwidth mobile communications alternative to cellular systems for multimedia applications
Peripheral:	Remote processor typically only performing immediate processing associated with a call (e.g. provision of dial tone)
PIC:	Point-In-Call; BellCore-defined call states
Pipeline:	A sequence of processing stages, with each stage performing part of the total job.
POTS:	Plain Old Telephone Service (basic connection service in a telephone network)
Processor:	Electronic circuitry which can execute stored instructions out of electronic memory
Program Module:	A segment of code which has to be executed on a single processor
Scaleability:	For a call processing system, the degree to which the number of processors in the multi-processor architecture can be increased (or decreased) to handle different numbers of users or services which require more processing

Shared Memory System:	Architecture in which (a portion of) the system Memory is common between one or more processors and is used as the medium for communication between different processors
Sub-Call:	Identifiable contiguous sequence of operations which can be identified during the processing of a call (corresponds directly here to a program module in the call processing software)
Task Assignment:	The decision of how to allocate of sub-tasks (program modules here) to processors
Task Scheduling:	The decision about when to allow execution of assigned tasks to begin
Telco:	Telephone Operating Company
Telecommunications Switch:	Hardware and software system which can set up and tear down telephone connections between users. Typically, the inputs and outputs are multiplexed signals (e.g. DS1 signals)
Telephone Service:	A particular function which can be performed using a telephone connected to the telephone network (e.g. Call forwarding, where incoming telephone calls can be sent on to another telephone number automatically)
Tightly-Coupled System:	Architecture with a high degree of interaction between processors; processors are linked closely together during operation
VLSI:	Very Large Scale Integration

Chapter 1

Introduction

The complexity of a telephone call is increasing with the advent of new features [1, 2, 3, 4] and, as a result, the processing power of telephone switch cores must also increase to handle such calls [5]. Since the Call Processing portion of the switch software occupies most of the CPU (60% to 85%) in a single-processor core system [6], it is the Call Processing design that is most critical to overall switch performance.

To meet increasing call processing requirements, telephone switch manufacturers are now considering parallel processing solutions for Call Processing architectures. Parallel processor architectures allow individual processor complexity to decrease, which often reduces cost and increases the reliability of the final system. Equally, these architectures allow existing powerful and expensive processors to work as part of a yet more powerful team of processors. Different multi-processor core architectures are possible, so it is necessary to understand the basic evolution of the telephone switch before attempting new designs.

1.1 A Brief History of Switch Architectures

A digital telephone switch is typically divided into a peripheral processing unit and a core processing unit [5, 7, 8]. The peripheral processing unit handles the local processing requirements associated with a call, while the core processing unit co-ordinates the establishment, maintenance and termination of all calls. This centralized core architecture has been used in telephone switches for the last twenty-five years [7] and is still being used in designs today.

Other work examining distributed or tightly-coupled architectures also appears promising [1, 9, 10, 11, 12, 13], but these architectures require significant changes to be made in the design and

organization of the switch software. This software is large and complex and has been optimized extensively; making the re-writing and re-testing of it an expensive and impractical option. For this reason, the work described here assumes a centralized core architecture, which allows re-use of a large portion of the existing code.

Although the basic switch design has remained constant, the architecture of the core processing unit itself has changed considerably over the past 25 years to satisfy demands for greater reliability, real-time performance and improved services [14, 15]

(1) In the earliest digital implementations, a single processor was used to replace the old relay-based analog switching core [7]. These new digital systems were smaller, more powerful and more reliable than their mechanical counterparts.

(2) Redundant processors were added to the switching core to improve reliability. The current specification for telephone switch availability in North America can be broadly stated as no more than 1 hour of downtime in a forty year period [16].

(3) More processors were added to the 'edges' or periphery of the switch to reduce the load on the central core and to maintain the real-time response of the system. The LSSGR standard [16] states that no more than 20% of calls should experience a delay exceeding 3 seconds during the High Day Busy Hour traffic period.

(4) More powerful processors were developed for the core, to take advantage of the progress in VLSI technology (i.e. increased density and speed) and advances in microprocessor design (e.g. the development of sophisticated caching and pipelining techniques). These new processors allowed switch cores to handle more telephone calls. A large switching office controls about 100,000 sending and receiving terminals (telephone sets), of which about 10,000 can be active simultaneously [17].

(5) Multiprocessor techniques were used to replace the active-standby core architectures. These multi-processor architectures ranged from centralized shared-memory and load-sharing architectures to distributed multi-computer configurations. However, most commercial Call Processing systems continued to use a centralized approach because of the challenge associated with synchronization and the increased real-time complexity of the software for a distributed system [18]. Also, the relatively slow growth in memory speed, when compared with the increased speeds of the processors limited the number of tightly-coupled shared-memory systems [19]. For example, the Northern Telecom DMS, the Alcatel E10, the Siemens EWS and the Ericsson AKE10 switches all use a centralized core architecture.

Table 1.1 shows the major multi-processor Call Processing systems which have been developed for commercial use.

The architectures of most commercial switches have evolved in a similar sequence and are now at the multi-processor stage. However, with the introduction of more advanced services there is an increase in the signalling required between the terminal and the switch during the call, as well as during the setup of the call [5]. Table 1.2 defines some newer services, which are now being offered.

These advanced services also interact with each other [47], further burdening the central core. For example, if customer A calls customer B, who is already on the phone, the Call Processing software must search its database to determine if B has subscribed to the 'Call Waiting' service. If so, a Call Waiting signal is sent to B to indicate that a new call is coming in. By 'flashing' the switch-hook, B can alternate between A and the original conversation. If B does not have the Call Waiting service, A receives a busy tone. However, if A subscribes to the 'Ring Again' service, the central office generates a call between A and B as soon as B finishes the current conversation.

Since a phone can call any other phone, the software process controlling a phone can interact with any other process controlling any other phone. These types of interactions cause application

Manufacturer	*Model*	*Deployment Period*	*Architecture Description*
Alcatel	E10 (and System 12)	1983-1996	Functionally distributed and loosely-coupled architecture. Call Processing functions are spread across networked subsystems, with load sharing amongst the processors within a subsystem. [20, 21, 22]
AT&T (Lucent)	5ESS	1982-1996	Distributed Call Processing architecture with clusters of active-standby processors and peripheral modules [23, 24]
Siemens	EWS-D DE5	1984-1996	Centralized architecture (dual processor; active-standby) with peripheral processing. The switch can be used in a more distributed configuration for large applications [25]
Northern Telecom	DMS-100	1979-1996	Centralized architecture (dual processor; active-standby) with peripheral processing [26, 27, 28]
Ericsson	AKE10 and AXE13	1980-1990 and 1980-1996	Centralized architecture (dual processor active-standby) with peripheral processing [29, 30]
NEC	NEAX61K	1979-1996	Tightly-coupled Multi-processors [31, 32, 33]
FUJITSU	FETEX-150	1980-1996	Load-sharing multiprocessor (tightly coupled) [34, 35]
NTT	D70	1981-1996	Function sharing and Load-sharing multiprocessor (tightly coupled) [36, 37, 38, 39]
Rockwell/ Wescom	580 DSS	1977-1990	Functionally distributed and loosely-coupled hierarchical multiprocessors [11, 40]
GTE	GTD-5 EAX	1979-1990	Centralized architecture (dual processor; active-standby) with peripheral processing (dual processor; also active-standby) [41, 42]
GEC	Mark II Multiprocessor	1973-1980	Centralized architecture [43]
HITACHI	HDX10	1980-1996	Hybrid System with load sharing possible in the centralized and peripheral portions of the switch [44, 45]
ITT	TCS-5	1977-1983	Load-sharing multiprocessor (tightly coupled) [9]
British Telecom	System X	1981-1992	Microprocessor modules interconnected in a ring; small-scale shared memory systems possible [8].
Phillips	TCP36	1978-1984	Centralized architecture with peripheral processing with shared memory and some functional distribution [46]

Table 1.1 Multiprocessor Approaches to Commercial Call Processing Systems

Service/ Feature Name	*Description of Feature*
Call Waiting (and Call Waiting Deluxe)	accepting another incoming call even though there is another call already in progress and then being able to multiplex between these two distinct calls
Ring Again	the ability to automatically re-try a busy number
Conferencing	more than two participants permitted in a multi-way conversation
Caller Identification Delivery	display of the caller's number or name on the receiver's telephone set and related services; these services require a handset with display capability
Calling ID Delivery Blocking	prevent delivery of a subscriber's ID by another subscriber with Caller Identification
Calling ID Delivery/Call Waiting	display of the ID of incoming calls which have been put on hold as well as the currently active call (see above for the definition of the Call Waiting service)
Calling Name Delivery	association of a telephone number with a subscriber's name and the display of that name
Calling Number Delivery	display of the caller's number
Anonymous Call Rejection	ability to prevent the reception of calls that do not have a name or number associated with them (e.g. subscribers that have Calling ID Delivery Blocking, for example)
Ring Control	customized rings for multi-party lines or multi-number lines with the same physical wire access
Visual Message Waiting Indication	visual indication that phone messages have been received (usually for a Telco-provided answering service or voice-mail)
Call Forwarding (Call Forwarding Don't answer, Call Forwarding on Busy)	automatic initiation of a call to another number or voice-mail service if the subscriber does not respond or cannot respond
Automatic Callback/ Last Call Return	allows initiation of an outgoing call to the same subscriber that last placed an incoming call (e.g. for cases when a subscriber 'just missed' picking up on time)
Automatic Recall	allows subscribers to scroll through the set of the last few calls which have been received
ISDN data services	For a detailed description of these services, see [48]
Higher bandwidth services	For a detailed description of these services, see [13] and [1]

Table 1.2 Non-POTS telephone services

complexity to multiply, since each telephony feature can require quite different messages and signals between the sender and receiver [2]. Indeed, with the introduction of mobile users [12], a

call may initiate on a particular switch and may migrate to a different switch (or switches) during the progress of the call.

To speed up the overall Call Processing operation and increase the system capacity for Call handling, advances in interconnect topologies must be used. Although single processor technology has also seen improvements, Call Processing applications cannot take advantage of all of these advances. For example, increased conditional branching in the code caused by feature interactions detracts from the use of instruction pipelining/caching. ALU pipelines are not cost-effective either, since data processing is minimized in most Call Processing applications. A third improvement, instruction/data caching, yields little speedup if there is limited instruction and data 'locality' (i.e. instructions or data are distributed widely across many pages of memory). This is common for Call Processing software which must co-ordinate complex features and their interactions.

1.2 Outline of the Thesis

Although several parallel architectures have been developed (Table 1.1) the architectures of most current commercial systems are not described in any detail in the literature. A more general study is attempted here, using Verilog simulations to model the behaviour of a Call Processing core on a new parallel architecture - the linear pipelined array.

Chapter 2 describes the Call Processing problem in detail. Chapter 3 describes parallel processor architectures suitable for Call Processing, and derives the general criteria necessary to evaluate the architectures. In subsequent chapters (4 and 5), one of these architectures, the linear pipelined array, is analyzed and then simulated to determine its suitability for Call Processing applications. The results of the analysis, discussed in Chapter 6, indicate that a heterogenous linear array can be used effectively for Call Processing applications and exhibits reasonable scaleability. Finally, conclusions and future work are described in Chapter 7.

1.3 Main Contributions

The main contributions of this thesis are:

- A review of the multi-processor architectures used for telecommunications call processing is summarized in Section 1.1,
- Detailed design criteria and requirements are generated for a telecommunications call processing system in Chapter 2 and Section 2.3.1,
- Comparison of three most promising architectures is made for call processing based on these design criteria in Section 4.1,
- A new scheme for Call Processing is proposed using a Linear Pipelined Array of heterogeneous processors (Chapter 4),
- Detailed simulations of this new approach are performed which demonstrate that:
  - dynamically selected schedules (based on information about job states) have better performance than static schedules,
  - mid-call delays, like those associated with user interaction, during processing can reduce performance significantly,
  - the architecture cannot scale to a large number of processors because of the requirement of maintaining balanced loads across processors, so parallel pipelines are needed for large systems.

Chapter 2

Call Processing

This chapter describes the Call Processing algorithm and discusses the models used for implementation. The basic requirements and design criteria for a Call Processing architecture are also presented and are used to evaluate the architectures discussed in subsequent chapters.

2.1 General Description

The basic function of Call Processing hardware and software is to establish a connection between two entities; a transmitting terminal (transmitter) and a receiving terminal (receiver). The transmitter and receiver are typically a telephone or modem. For long-distance calls which traverse more than one switch, intermediate switches behave like terminals themselves, originating and terminating connections to an adjacent switch.

In a circuit-switched network, the connection is a hard physical path, such as a dedicated wire set up to carry traffic between the terminals for the duration of the call or message. In contrast, newer fast packet-switched networks also permit logical connections, where data is split up into blocks and routed using 'labels' which contain address information for either the sender or the receiver. The discussion of Call Processing in subsequent chapters pertains mostly to circuit-switched telephone networks, but many of the concepts and operations discussed are also relevant to these newer packet-switched networks.

The basic steps associated with Call Processing for a POTS (Plain Old Telephone Service) call are described in the following paragraphs and illustrated in Fig. 2.1:

(1) **Incoming call detection**: The Call Processor detects the initiation of a call from the sender. In a telephone network, this involves detecting the 'off-hook' signal from the

sender.

(2) **Digit reception**: The signalling scheme used by the sender is determined and the digits are collected.

(3) **Digit translation**: The collected digits are analyzed and used to determine the correct receiver.

(4) **Routing**: The sender and the receiver are determined and the best available path through the network is selected.

(5) **Connection**: The appropriate hardware and software are provisioned to set up the selected path.

(6) **Receiver notification**: The receiver is informed that the sender has initiated a call (e.g. ringing). Some feedback to this effect is also conveyed to the sender (e.g. audible ringback tone).

(7) **Receiver acknowledgment detection**: The Call Processor waits for a response from the receiver (e.g. off-hook signal) and then logs the time of the 'start' of the call, for billing purposes.

(8) **Disconnect detection**: The Call Processor waits for either the sender or the receiver to disconnect (e.g. on-hook signal in a telephone network) and then logs the time of the 'end' of the call, for billing purposes. The established path is removed and both the sender and the receiver are returned to the idle state.

Note that this basic model is complicated by the introduction of more advanced services and different types and mixes of services as described in Chapter 1. Once these new services are introduced, state changes in a call may cause state changes in other calls which are in progress.

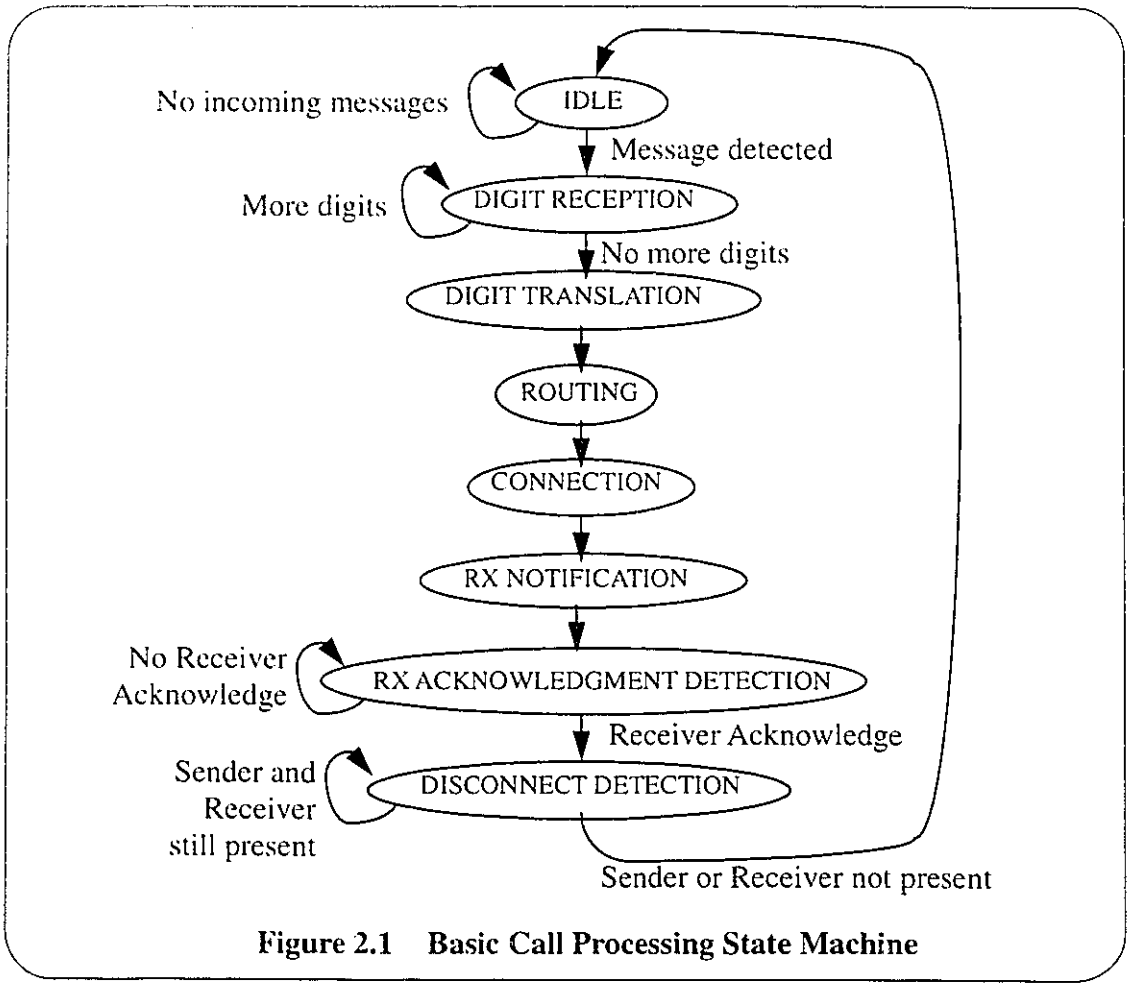


Figure 2.1 Basic Call Processing State Machine

2.2 The Call Processing Algorithm

The basic Call Processing state machine can be further broken down [49, 50] and modelled as separate sequences of actions for both the calling party and the called party. These are termed the Originating and Terminating half-call models, respectively, and are described in Appendix A.

The Originating half-call model responds to a call origination request and sets up a path (for signalling and transmission of data) to be added into a service switching point (typically a switch). The Terminating Basic Call model handles the setup for a path to be dropped out of a service switching point. By modelling a call as two halves, the processing associated with the calling party and the called party can be treated independently.

For a given call, there can be multiple service switching points which have to be connected properly in series to make the desired connection between network users. In this case, each switch along the route maintains origination and termination half-call models. Although some information needs to be exchanged between the two models during call setup, this functional decoupling simplifies the Call Processing software and has been adopted by most switch vendors. These models are the basis for the simulations described in Chapter 5.

2.3 Requirements

The fundamental requirement of a Call Processing architecture is that it must support an efficient implementation of the half-call models described above. A good description of the general requirements for a Telecommunications Call Processing System is given in [51 - 65]. Other requirements for a Call Processing architecture are:

(R1)The architecture should be scaleable, allowing different numbers of users or mixes of traffic type to be accommodated by adding more processors. As more processors are added, the overall processing power of the system should scale proportionately.

(R2)The system should perform well under overload conditions. This results in the following sub-requirements:

- Under overload conditions, some delays in dial tone to the user are allowed, but no other delays may occur. Once a call is accepted, a minimum level of service must be maintained. This is especially true under overload conditions, where demand exceeds the Call Processing capacity.

- The rejection of call attempts should be 'fair' under overload conditions, for both calling and called parties. That is, users should have an equal probability of receiving initial service.

(R3)The system should be fault-tolerant [8, 66, 67]. This results in the following sub-requirements:

• Single faults in any of the components (e.g. processors, memory, communication links, c.t.c.) should be tolerated, without degrading performance beyond the system specifications [16].

• Error conditions should be handled 'gracefully' without requiring manual intervention or degrading the overall reliability of the switch.

• The Call Processing architecture should be repairable in operation without disruption of service. Software should also be upgradeable when the system is in service.

(R4)The system should be reliable. This results in the following sub-requirements:

• All persistent requests for service must eventually be honoured.

• The probability of call connection should be high (high system availability).

• The probability of call cutoff should be low. Single failures in the Call Processing system should not result in significant loss of service to more than 100 users. This qualifies as an 'incident', which must be reported to the FCC authority as an outage within 24 hours of occurrence.

(R5)The architecture should perform well and overall Call Processing capacity should be maximized. This is usually measured in terms of the number of calls, but the complexity of a given call is increasing more rapidly than the number of calls in the current North American market [5]. Thus, a more reasonable measure includes both the quantity and the type of calls being processed.

• The response time to the user (e.g. dial tone) should be less than 3 seconds [16], as there is evidence that users give up or dial without waiting for dial tone after this time [15]. This user behaviour can escalate an overloaded switch condition rapidly, because of the processing required to support the increased number of call attempts by frustrated users

[55]. There is a 'hysteresis' effect for system recovery, such that the system will only recover once the total of the offered load and the customer re-tries is equal to the total Call Processing capacity of the system [63].

• The overhead associated with Call Processing tasks should be minimized (e.g. scheduling; storage of call states during lulls in activity) since the Call Processing task is characterized by large real-time gaps in between bursts of processing.

• The amount of processing wasted on calls which are not going to be completed should be minimized. This means that:

(A) Calls in progress should not be dropped in favour of newer calls,

(B) Calls which have already used a substantial amount of system resources should be completed; this avoids many of the 'thrashing' failure scenarios and

(C) Newer calls should be favoured over misdialled calls.

(R6)The architecture should support varying degrees of user interaction efficiently during the processing of a call. This user interaction can be for user service or service option selection/termination, or user signalling (required during specific services).

(R7)The software should be hardware-independent and modular. The introduction of new features and processor platforms should not increase the complexity of the software architecture. It is assumed that all architectures are software-intensive; specialized hardware architectures are not practical, because of the frequent changes in telecommunication switch features and their interactions [47].

2.3.1 Comparison Criteria

The following criteria are used in Chapter 4.1 to evaluate the three different architecture types described in Chapter 3. These criteria are derived from the requirements listed in Section 2.3:

- Scaleability: The ability of the architecture size to be varied to accommodate different problem sizes.
- Performance: The throughput and response time for calls, especially under overload conditions.
- Software Complexity: The modularity and maintainability of the software blocks.
- System Reliability and availability: The likelihood of system failures and the ability of the system to tolerate these failures, if they occur.
- Support for User Interaction: The ability to support variable delays associated with the gathering of user data during the processing of a call.

2.4 Summary

In this chapter, the basic Call Processing functions have been described and the design requirements for a Telecommunications Call Processing system have been given. These requirements have been grouped together into the design criteria which are used to evaluate different Call Processing architectures in later chapters.

Chapter 3

Review of Multiprocessor Architectures for Call Processing

This chapter describes several multi-processor architectures which have been used for Call Processing and gives the relative advantages and disadvantages of each. The architectures differ in one or more of the following:

- interconnection topology,
- messaging method,
- assignment of program modules/sub-calls* to processors (fixed or variable[68]).

3.1 Linear Pipelined Arrays

A linear pipelined array consists of an array of processors, each with its own local memory (Fig. 3.1). Each processor has one upstream and one downstream neighbour, with the exception of the first processor and the last processor in the array which have only a downstream and an upstream neighbour, respectively. The processors in the array can be identical (homogenous architecture) or not (heterogenous architecture). Simple point-to-point bidirectional links connect processing units to their immediate neighbours. This interconnection topology minimizes the complexity of the hardware and the required supporting software, although it reduces the flexibility of the communications structure.

Messaging between adjacent processors is simple and direct, but messaging between non-adjacent processors requires a 'pass-through' mechanism. For example, to broadcast a message to n downstream processors, n-1 'pass-through' operations are required in addition to the reception of

* The terms 'module' and 'sub-call' are used interchangeably here.

the message at n sites. This makes the implementation of algorithms requiring communication between many processors, such as broadcasting, less suitable for the linear pipelined array because of the lost execution time during the 'pass-through' of messages. Memory may be shared between processors, but this reduces processor independence and introduces performance bottlenecks.

The linear pipelined array is a centralized architectural approach for Call Processing. The operation of a processor in the array is highly dependent on the operation of upstream processors in the array. However, it is possible to organize the array in different ways to execute the Call Processing function:

- processors can handle the same stage of different calls,
- processors can handle different stages of different calls.

The first organization is intuitively the most suitable for the linear array and is developed and analyzed further in subsequent chapters.

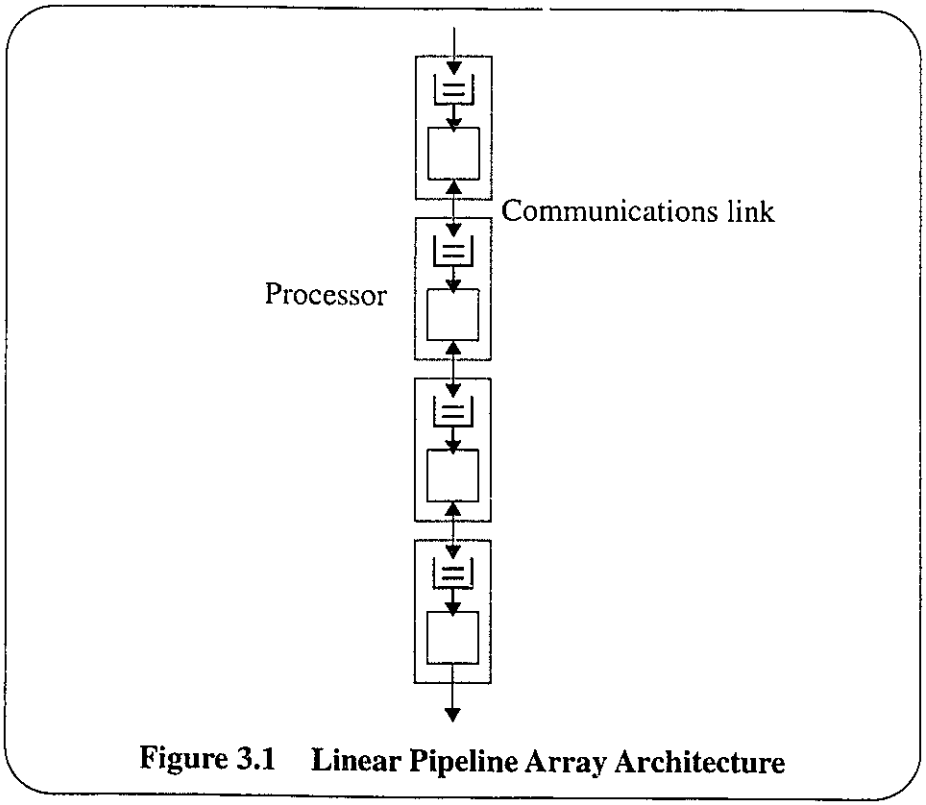


Figure 3.1 Linear Pipeline Array Architecture

3.2 Load Sharing and Functional Partitioning

The Load Sharing architecture (Fig. 3.2) consists of a 'pool' of processors which are all connected to a common demultiplexor (load distribution source) and multiplexor (output collation mechanism). The multiplexor and demultiplexor may be processors themselves, or may be implemented using shared memory [69].

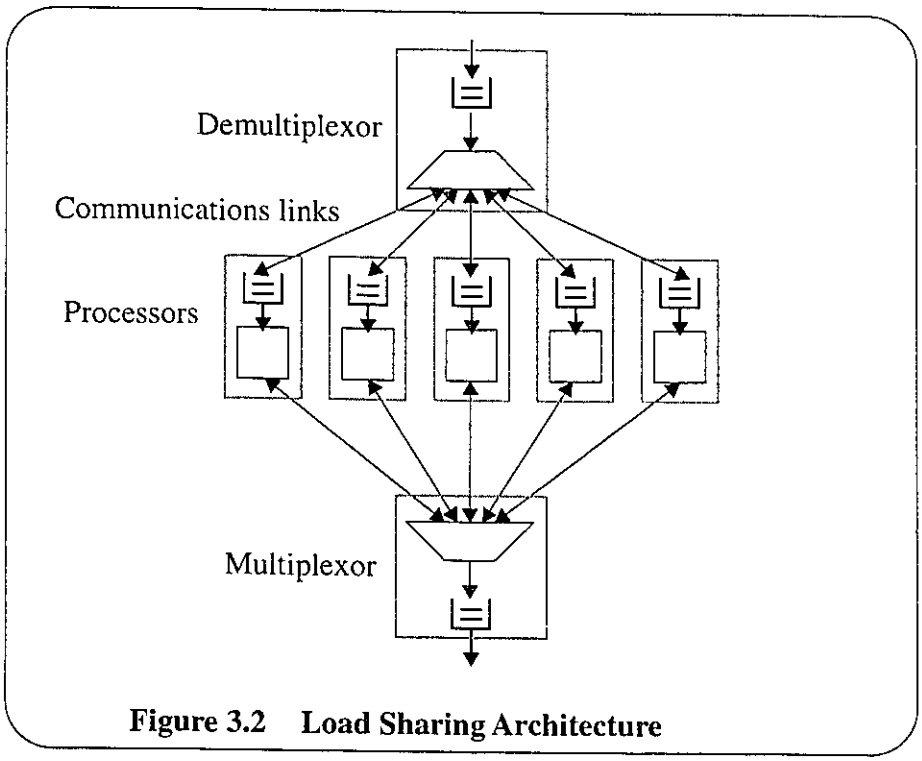


Figure 3.2 Load Sharing Architecture

In the Load Sharing architecture, each processor executes all of the program modules for the Call Processing function, so messaging between processors is unnecessary for the processing of a single call. The processors in the architecture are usually assumed to be identical for analytical simplicity [70, 71], making the architecture homogenous. There is no specialization assumed in such architectures; each processor performs the entire Call Processing task.

A functionally partitioned architecture has the same basic structure as the load sharing architecture, but has no restrictions on processor homogeneity. Each Processor is allocated work by the demultiplexor for which it is the most suited. For example, the Ericsson switch

architecture can use specialized processors for different Call Processing features [29, 30]. In a functionally partitioned architecture, the processors can be connected in different ways, but the assignment of work to processors is still fixed.

3.3 Distributed Architectures

A distributed architecture uses decentralized processors, each with their own local memory, connected through networks which perform the message and data routing functions. By distributing the network call control function, an 'Intelligent Network' is constructed [7, 72]. Here, Call Processing is no longer concentrated at the local switch, but is distributed throughout the network. With this approach, switching systems can operate in conjunction with a variety of other network databases and systems, making a variety of new services possible.

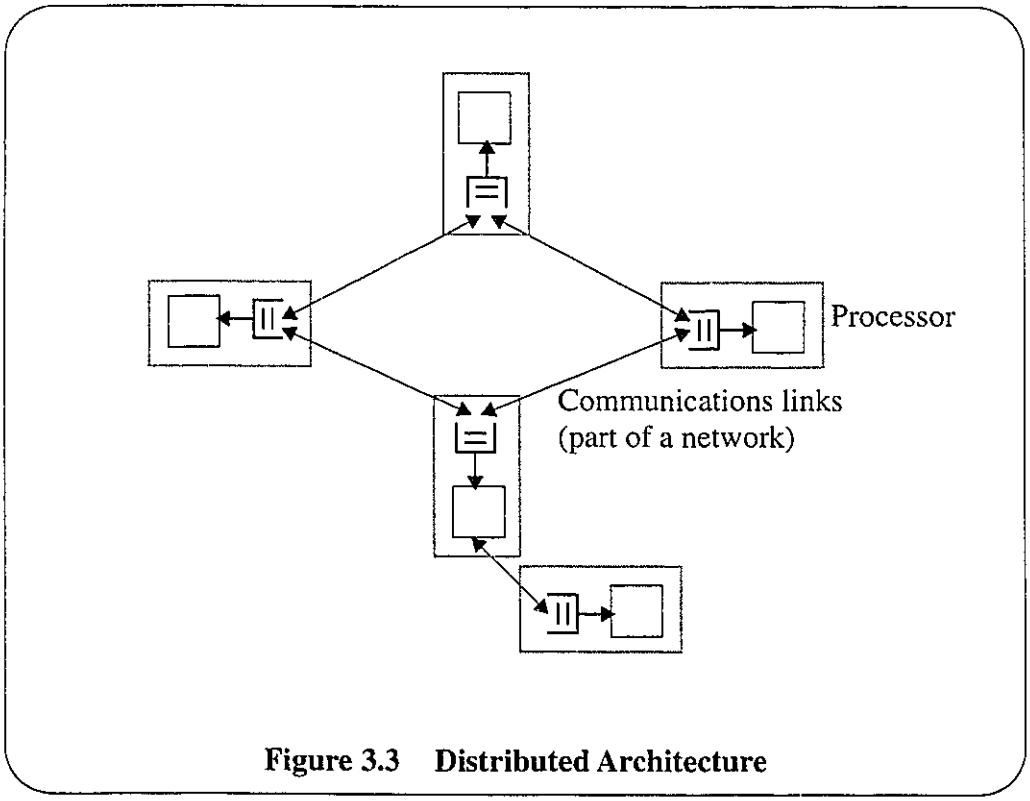


Figure 3.3 Distributed Architecture

The complexity of the distributed architecture and the difficulties of managing distributed networks make implementation harder than for more centralized architectures like the Linear

Array and Load Sharing architectures. Performance estimation and Engineering is also more difficult. The most successful distributed architecture is the AT&T 5ESS switch. Here, performance is guaranteed by providing excessive switch processing capacity, rather than by attempting to solve the efficiency issues associated with such a distributed system.

3.4 Summary

In this chapter, three different multiprocessor architectures for Call Processing have been introduced:

- Linear Pipelined Array - a novel approach.
- Load Sharing architecture - a common approach.
- Distributed architecture - another promising approach.

All of these architectures can be used for Telecommunications Call Processing. Using the design criteria from Section 2, an evaluation and analysis of the three different architectures for Call Processing is performed in Section 4.

Chapter 4

The Linear Processing Array

In the previous chapter, the linear pipelined array was shown to have some characteristics which made it suitable for the Call Processing application. From Table 1.1, it can be seen that this approach has not yet been used by any of the main switch vendors; the Load Sharing architecture is a much more common multi-processor approach to Call Processing. Therefore, the linear pipelined array is now evaluated and compared with the Load Sharing architecture for implementation of the Call Processing application, based on the criteria generated in Section 2.3.1. The objective of the comparison is to determine whether the novel approach of using a Linear Array for Call Processing is justified or not.

4.1 Architecture Comparison

The relative advantages of all of the architectures described in the previous chapters are summarized in Table 4.1. The architectures become more flexible towards the right side of the table. However, this flexibility is at the expense of increased complexity, which makes analysis and prediction of performance more difficult. It can be seen that the Linear Array is one of the simpler architectures to analyze*.

---

* It is also interesting to note that communications speed, which is defined as the average rate (after protocol delays and message transmission and reception) at which messages can be sent and received between adjacent processing units has not increased at the same rate as processing speed in multiprocessor systems [19]. Therefore, an architecture like the linear array, which attempts to minimize interconnections and the accompanying messaging, is expected to perform well with current processor technology.

	Linear Pipelined Array	*Load Sharing/ Functionally Partitioned*	*Distributed*
Performance	✔ Simple Interconnect ✔ Scheduling problem has a known solution [73] ✔ Response time is predictable because of tightly-coupled nature ✖ Execution bottlenecks may occur, causing processors to starve for input ✔ Stable under overload conditions	✔ Centralized/simple communication with low overhead ✔ Response time is quite predictable ✔ Stable under overload conditions ✖ Difficult to take advantage of processor specialization without incurring inter-processor delays ✖ Memory bottlenecks and deadlocks may occur	✔ Good for 'remote' features, where only local information is required ✖ Network communications bottlenecks may occur ✖ Difficult to take advantage of processor specialization without incurring inter-processor delays ✖ Difficult to predict/ guarantee delays and thus service quality ✖ Unpredictable for overload conditions, because of communications network blocking
Scaleability	✖ Limited scaleability	✔ Good scaleability	✔ Good scaleability
Software Complexity	✖ Tightly-coupled and specialized or restrictive software	✔ Uniform and relatively conventional Software	✔ Software size limited at each site ✔ Uniform and relatively conventional Software ✖ Complex asynchronous inter-processor communication
System Reliability/ Availability	✔ Average; linear interconnection lends itself to a ring-style protection mechanism[a]	✖ Sharing of memory is the reliability issue	✔ Excellent.
User Interaction	✔ Average; accessing user data requires a complete pass through the array	✔ Good; requires a multiplexing/ demultiplexing mechanism for user data	✔ Excellent; Processors are responsible for a particular set of users and the data collection for those users

Table 4.1 Comparison of different Multiprocessor Approaches to Call Processing

a. Current technology and protocols exist for efficient and fast switching of traffic between elements which are connected in a linear fashion [74]. To protect against the loss of functionality in a particular processor, it is necessary to duplicate capabilities between neighbouring processors (at worst, doubling the number of processors in the array for full sparing of each processor functionality at a neighbouring site).

Distributed architecture applications often require specialized software which has a very high initial cost [75]. Existing switching software has been designed to process calls for large and complex hierarchical switching networks [76] and the re-use of this software is a requirement for many manufacturers. Even if a software redesign is possible, another challenge for distributed architectures is to deploy hardware and software which can still be used as new features are conceived and deployed. If this is not done, the introduction of new services requires costly and complex network upgrades and re-testing. The distributed approach is included here for completeness, since it has been used in some commercial Call Processing systems [24, 77, 78], but is, however, not examined further because of these significant disadvantages.

The key criteria of Performance and Scaleability are now refined and used to make a more detailed comparison between the proposed Linear Array approach and the more ubiquitous Load Sharing approach. The motivation for most multi-processor architecture study is the reduction in the execution time of programs. For the purposes of this work, the program execution time is the time required to:

- load the relevant call context data into memory,
- execute the stored instructions for the call state and
- store the resulting changes in the call context data.

The execution time is affected by:

- the processing speed of each processing unit.
- the memory access and I/O speed between the processing unit and its peripherals.
- the messaging speed between processing units.
- overhead associated with any of the above, including interrupt service routine overhead.

For a Call Processing application, the most significant effect of reduced program execution time is the increase in call throughput (or number of calls which can be handled in a given period of time). To balance the study of performance, it is also necessary to consider the cost of the system.

Assuming that processor cost is inversely proportional to processor instruction execution time, a cost factor is defined as the inverse of the sum of execution times for all of the software modules that make up a call summed over all processors. Therefore, the following sections use throughput as the main comparison criteria for different architectures.

4.1.1 Notation

The remainder of this chapter uses the following definitions adapted from the work of Bokhari *et al.* [79, 80, 81] on the performance analysis of linear arrays. A 'sub-chain' is defined to be a contiguous sequence of program modules, which will be assigned to a single processor (processors are numbered from 1 to p):

w_i execution time of module i (numbered from 1 to m)

c_i communication time between module i and module $i+1$ ($c_0$and $c_m$are the communication times with the outside world for module 1 and m, respectively.

$W_{p,s,t}$ load on processor p if a subset of the m modules, sub-chain $s...t$, is assigned to it. This is referred to as the weight of sub-chain p (the subchain assigned to processor p).

$$\Omega_{p,s,t} = \sum_{i=s}^{t} w_i + c_t + c_{s-1}$$

W load on a processor if all m modules are assigned to it.

$$W = \Omega_{p,1,m} = \sum_{i=1}^{m} w_i + c_0 + c_m$$

$\tau(p)$ a vector of length n that specifies the particular sub-chains of modules assigned to each processor, specifying the 'handover' points of the subchains for each processor. Processor p is defined to have the sub-chain $\tau(p-1) + 1$ to $\tau(p)$ assigned to it, with $\tau(0) \equiv 0$. This is also called a partition.

bottleneck processor/ sub-chain:

for a given $\tau(p)$ the bottleneck processor is the processor/ sub-chain with weight:

$$\max_p \{\Omega_{p,\tau(p-1)+1,\tau(p)}\}$$

$\omega(\tau(p))$*weight of a partition* = weight of its bottleneck processor. The optimal partition is the $\tau(p)$ for which the weight $\omega(\tau(p))$ is minimum.

4.1.2 Throughput

Consider a system of n processors, whose task is to process Call Processing jobs, where each call requires m modules of software. The objective is to maximize the call throughput (the number of calls processed in a given time) for a given number of processors.

Specifically, consider a heterogenous linear pipelined array and the common homogenous Load Sharing architecture discussed in Chapter 3. Other than the bottleneck associated with job distribution to the processors and output collation, the Load Sharing architecture scales well. Scheduling of identical Call Processing jobs can be performed using a simple round-robin strategy. The performance of a linear pipelined array which is constrained to be constructed of the same homogenous processing elements, will always be worse or at most equal to that of a Load Sharing architecture made up of the same processors, because of the increased messaging needed between processors in the array. This is shown as follows:

The maximum throughput for the pipeline architecture is given by:

$$\text{throughput (calls/time unit)} = \frac{1}{\omega(\tau(p))}$$

The maximum throughput for the load sharing architecture, assuming there are no bottlenecks at

the multiplexor and demultiplexor, is given by:

$$\text{throughput (calls/time unit)} = n \times \text{throughput for a single processor}$$
$$= n \times \frac{1}{\sum_{i=1}^{m} w_i + c_0 + c_m}$$

For the pipelined architecture to achieve a higher throughput than the load sharing architecture, the pipeline must have a maximum (bottleneck) program module execution time which is smaller than the average execution time for the Load Sharing architecture.

Theorem 1: The throughput for the Load Sharing Architecture will always equal or exceed that of the pipeline if the same number and type of processors are used for both. The proof is simple and is as follows:

Proof:

$$\text{Throughput}_{\text{pipeline}} = \frac{1}{\omega(\tau(p))}$$

$$\text{Throughput}_{\text{Shared Memory}} = \frac{n}{\sum_{i=1}^{m} w_i + c_0 + c_m}$$

Show that, for any given set of w_i's,

$$W = \frac{n}{\sum_{i=1}^{m} w_i + c_0 + c_m} \geq \frac{1}{\omega(\tau(p))}$$

Expanding terms and re-arranging, this reduces to:

$$\max_p \{\Omega_{p,\tau\langle p-1\rangle+1,\tau\langle p\rangle}\} \geq \frac{\sum_{i=1}^{m} w_i + c_0 + c_m}{n}$$

$$\max_p \{\sum_{i=\tau\langle p-1\rangle}^{\tau\langle p\rangle} (w_i) + c_{\tau\langle p\rangle} + c_{\tau\langle p\rangle-1}\} \geq \frac{\sum_{i=1}^{m} w_i}{n} + \frac{c_0}{n} + \frac{c_m}{n}$$

By definition, pipeline partitions are non-overlapping and include all m modules. Also, the communications costs associated with module 1 and module m, which are c_0 and c_m, respectively, must be included. The minimum value for the maximum of a series of numbers is equal to the

mean of that series (occurs when all numbers are equal to the mean). Therefore, the minimum value of the left side of the inequality can be no smaller than:

$$\frac{\sum_{i=1}^{m} w_i + c_0 + c_m + c_{\tau\langle 2\rangle} + c_{\tau\langle m-1\rangle} + \sum_{p=3}^{m-2} \langle c_{\tau\langle p\rangle} + c_{\tau\langle p\rangle - 1}\rangle}{n}$$

which, even for $c_{\tau(i)}$ with a value of zero, will always be greater than or equal to:

$$\frac{\sum_{i=1}^{m} w_i}{n} + \frac{c_0}{n} + \frac{c_m}{n}$$

However, if the processors are allowed to differ, the architecture is heterogenous and the throughput of the linear array may exceed that of the Load Sharing architecture. Since the processors in the linear array do not process **all** of the modules that make up a call, the throughput of the linear array will be better than that of the load sharing architecture, whenever the reciprocal of the maximum delay, taken over all stages in the linear array, is greater than the average of the throughputs for all of the processors in the Load Sharing architecture.

A simple numerical example is useful to show how sensitive the pipeline throughput can be to changes in the bottleneck module performance. Assume three different processor types (types *A*, *B* and *C*) are used to create a 5-processor array, which must run 5 program modules. Let the processor/module execution time array, $W_{i,j}$ be as shown in Fig. 4.1. Each entry in the array gives the execution time of program module *i* on processor *j*. For example, the fourth module takes 15 time units to execute on the second and third processors, which are both of type *B*. Bold face is used to show which executions are preferred and italics are used to show the bottleneck assignment. If a 5-processor pipeline with an optimal schedule is constructed, the throughput is limited by the first processor, giving a best case of 1/13 (0.077) jobs per time unit.

Throughput for a 5-processor Load Sharing architecture is optimized using all of type *A* processors, giving a throughput of 5/59 (0.085) jobs per time unit, which out-performs the linear array. In fact, in this first example, load sharing using any processor type results in equal or better performance than with the linear array (Table 4.2).

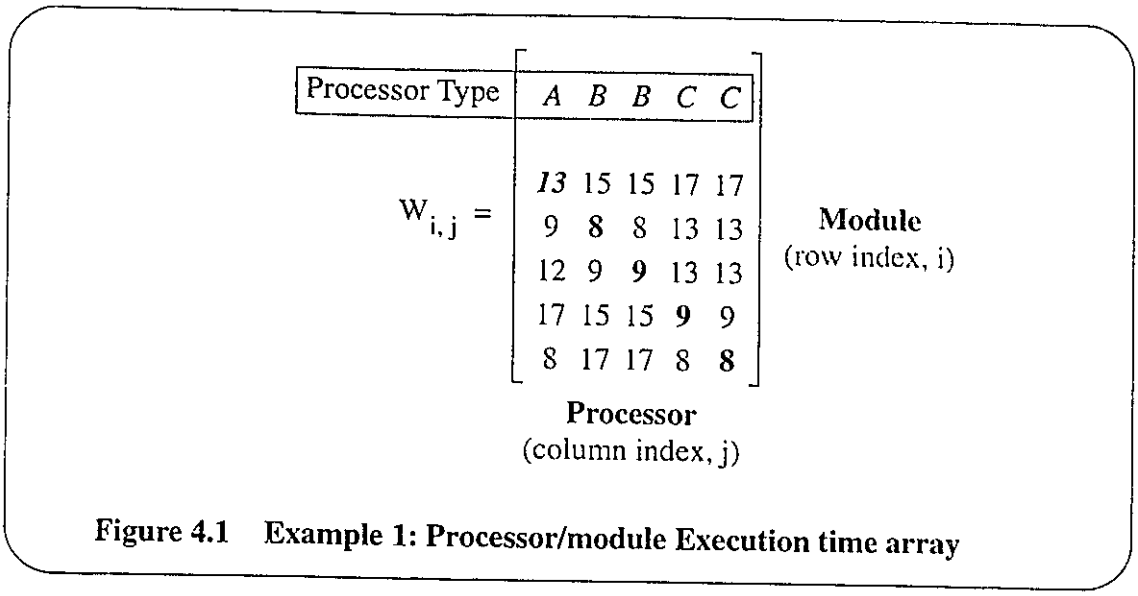


Figure 4.1 Example 1: Processor/module Execution time array

Processor type	*Load Sharing Type A*	*Load Sharing Type B*	*Load Sharing Type C*	*Linear Array (made with Types: A, B, B, C, C)*
Throughput	5/59	5/64	5/60	1/13
Throughput ratio (relative to the Linear Array)	1.10	1.02	1.08	≡ 1

Table 4.2 Example 1: Relative Throughput Comparison for a Load Sharing Architecture

Now adjust entry $w_{1,1}$ to be equal to 9, resulting in the new processor/module execution time array, $W_{i,j}$, as shown in Fig. 4.3 and the relative Load Sharing throughput is as shown in Table 4.3. With this change, the throughput of the pipeline always exceeds that of the three possible homogenous Load Sharing architectures (Table 4.3).

Processor type	*Type A*	*Type B*	*Type C*	*Linear Array (made with all three Types)*
Throughput	5/55	5/64	5/60	1/9
Throughput ratio (relative to the Linear Array)	0.82	0.70	0.75	≡ 1

Table 4.3 Example 2: Relative Throughput Comparison for a Load Sharing Architecture

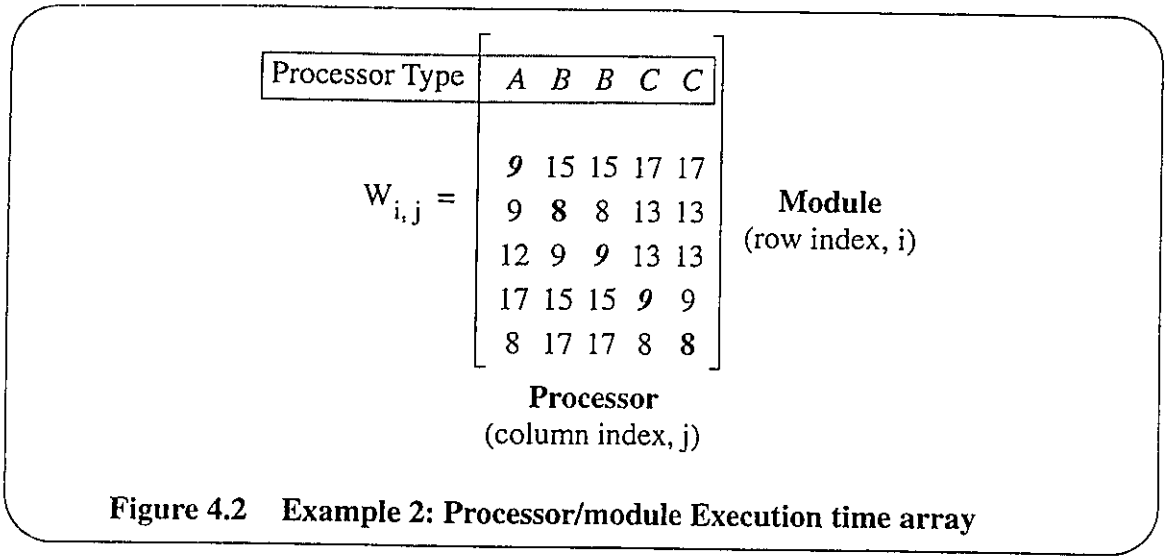


Figure 4.2 Example 2: Processor/module Execution time array

These examples illustrate the possible benefits of using a linear array, but also highlight the sensitivity of the linear array to changes in the processing time of the bottleneck program module. A major factor in this processing time is the cost of inter-module communications. In general, if this cost is high, the Load Sharing architecture will out-perform the linear array. The opposite is true for high inter-job or inter-Call costs, because in this case the multiplexing/demultiplexing requirements (Fig. 3.2) are a bottleneck for the Load Sharing architecture. Although the linear array is not fully interconnected, only certain communication paths are precluded, making it a better choice. The processors closest to the ends of the array have the greatest communication distance between themselves and other processors; communication between these processors is blocked most often by other communication paths.

4.1.3 Cost

Table 4.4 shows the cost factors for the three processor types of Section 4.1.2. A load Sharing architecture constructed solely of Type *A* processors would have a cost factor of 1/295 (compared with a linear array cost factor of 1/307). Improving the throughput of the architecture should come at a cost which is proportional.

Processor Type	*Total Sum of module Execution Times*	*Cost factor for Five Processors*
A	13+9+12+17+8=59	1/(5*59)=1/295
B	15+8+9+15+17=64	1/(5*64)=1/320
C	17+13+13+9+8=60	1/(5*60)=1/300

Table 4.4 Cost Factors for the Three Processor Types of Example 1

Table 4.2 shows the cost factors (and also the throughput factors) for the three possible homogenous Load Sharing architectures and the linear array for example 1. The homogenous Load Sharing architecture, with type *B* processors, has superior performance and cost factors to those of the best possible pipelined linear array for the specified processor/module execution time array. Indeed, for all processor types, the throughput increase is economical in the sense that the relative cost factor is not as high as the relative throughput factor.

Processor type	*Load Sharing Type A*	*Load Sharing Type B*	*Load Sharing Type C*	*Linear Array (made with Types: A, B, B, C, C)*
Throughput	5/59	5/64	5/60	1/13
Cost factor	1/295	1/320	1/300	1/307
Throughput ratio (relative to the Linear Array)	1.10	1.02	1.08	≡ 1
Cost factor ratio (relative to the Linear Array)	1.04	0.96	1.02	≡ 1

Table 4.5 Example 1: Throughput/ Cost factor comparison for a Load Sharing Architecture

For example 2, the cost factors are shown in Table 4.4. The cost is higher than a linear array, except for the case of a Load Sharing architecture constructed solely with processors of type *B*. However, the throughput is worse in this case than for the linear array. In general, the throughput decrease of the Load Sharing architecture is not offset by a corresponding cost decrease (Table 4.3).

Processor Type	*Total Sum of module Execution Times*	*Cost factor for Five Processors*
A	9+9+12+17+8=55	1/(5*59)=1/275
B	15+8+9+15+17=64	1/(5*64)=1/320
C	17+13+13+9+8=60	1/(5*60)=1/300

Table 4.6 Cost Factors for the Three Processor Types of Example 2

Processor type	*Type A*	*Type B*	*Type C*	*Linear Array (made with all three Types)*
Throughput	5/55	5/64	5/60	1/9
Cost factor	1/275	1/320	1/300	1/303
Throughput ratio (relative to the Linear Array)	0.82	0.70	0.75	≡ 1
Cost factor ratio (relative to the Linear Array)	1.10	0.95	1.01	≡ 1

Table 4.7 Example 2: Throughput/ Cost factor comparison for a Load Sharing Architecture

4.1.4 Scaleability

The ability to decrease overall execution time by adding more processing units is a fundamental attribute of a multi-processor system and is termed the scaleability of the system. If performance does not improve proportionately as more processors are added, the architecture is limited to certain sizes of problems. In practice, proportional (linear) speedup, as processors are added to a system, is seldom achieved and all architectures have a scaling limit [82, 19].

The linear array does not scale as well or as easily as the Load Sharing architecture. This can be seen by continuing the example. If more processors are added, Fig. 4.3, the linear array throughput remains fixed at 1/9 jobs per time unit and adding more processors only increases system cost.

To solve this problem, a pass-through mechanism which allows load to be shared within the array can be used [83]. In this scheme, the same program modules can be executed on different

processors to allow the load to be distributed more evenly; various scheduling schemes for this are examined in Section 6.1.

$$W_{i,j} = \begin{matrix} A & B & B & C & C \end{matrix}$$

$$\begin{bmatrix} 9 & 15 & 15 & 17 & 17 \\ 9 & 8 & 8 & 13 & 13 \\ 12 & 9 & 9 & 13 & 13 \\ 17 & 15 & 15 & 9 & 9 \\ 8 & 17 & 17 & 8 & 8 \end{bmatrix} \Rightarrow \begin{bmatrix} 9 & 9 & 15 & 15 & 17 & 17 \\ 9 & 9 & 8 & 8 & 13 & 13 \\ 12 & 12 & 9 & 9 & 13 & 13 \\ 17 & 17 & 15 & 15 & 9 & 9 \\ 8 & 8 & 17 & 17 & 8 & 8 \end{bmatrix} \Rightarrow \begin{bmatrix} 9 & 9 & 15 & 15 & 15 & 17 & 17 \\ 9 & 9 & 8 & 8 & 8 & 13 & 13 \\ 12 & 12 & 9 & 9 & 9 & 13 & 13 \\ 17 & 17 & 15 & 15 & 15 & 9 & 9 \\ 8 & 8 & 17 & 17 & 17 & 8 & 8 \end{bmatrix} \Rightarrow$$

$$\begin{matrix} & & & & & & & & & A & A & B & B & B & B & C & C & C & C \end{matrix}$$

$$\begin{bmatrix} 9 & 9 & 15 & 15 & 15 & 17 & 17 & 17 \\ 9 & 9 & 8 & 8 & 8 & 13 & 13 & 13 \\ 12 & 12 & 9 & 9 & 9 & 13 & 13 & 13 \\ 17 & 17 & 15 & 15 & 15 & 9 & 9 & 9 \\ 8 & 8 & 17 & 17 & 17 & 8 & 8 & 8 \end{bmatrix} \Rightarrow \begin{bmatrix} 9 & 9 & 15 & 15 & 15 & 15 & 17 & 17 & 17 \\ 9 & 9 & 8 & 8 & 8 & 8 & 13 & 13 & 13 \\ 12 & 12 & 9 & 9 & 9 & 9 & 13 & 13 & 13 \\ 17 & 17 & 15 & 15 & 15 & 15 & 9 & 9 & 9 \\ 8 & 8 & 17 & 17 & 17 & 17 & 8 & 8 & 8 \end{bmatrix} \Rightarrow \begin{bmatrix} 9 & 9 & 15 & 15 & 15 & 15 & 17 & 17 & 17 & 17 \\ 9 & 9 & 8 & 8 & 8 & 8 & 13 & 13 & 13 & 13 \\ 12 & 12 & 9 & 9 & 9 & 9 & 13 & 13 & 13 & 13 \\ 17 & 17 & 15 & 15 & 15 & 15 & 9 & 9 & 9 & 9 \\ 8 & 8 & 17 & 17 & 17 & 17 & 8 & 8 & 8 & 8 \end{bmatrix}$$

Figure 4.3 Example 3: Processor/module Execution time array (p=5,6,7,8,9,10)

Alternatively, multiple pipelines can be used [84, 85]. If multiple pipelines are used (Fig. 4.4) for groups of consecutive program modules (partitions), the different tasks do not interfere with each other. This is similar to the 'pipeline farm' proposed in [86] or the 'partitionable/parallel pipeline' proposed in [87]. Unfortunately, there is the same performance bottleneck as with the load sharing architecture discussed in Section 3.2 since a multiplexor and a demultiplexor are required.

Judicious introduction of extra delays can increase pipeline throughput [88, 89, 90]. Processors are added to provide a processing capacity 'buffer' which can be used as a reserve for unpredictable changes in the workload intensity or sub-call mix, which is typical for Telephone switches [76]. Note that this is not the same effect as the introduction of static delays to improve the basic throughput of a pipeline which has a fixed reservation table [82].

Performance may be improved by subsequent addition of processors, even though there are more processors than program modules. Obviously, processor skipping is required for this case. The scheduling decision of which processor to leave idle can be dynamic, where different processors

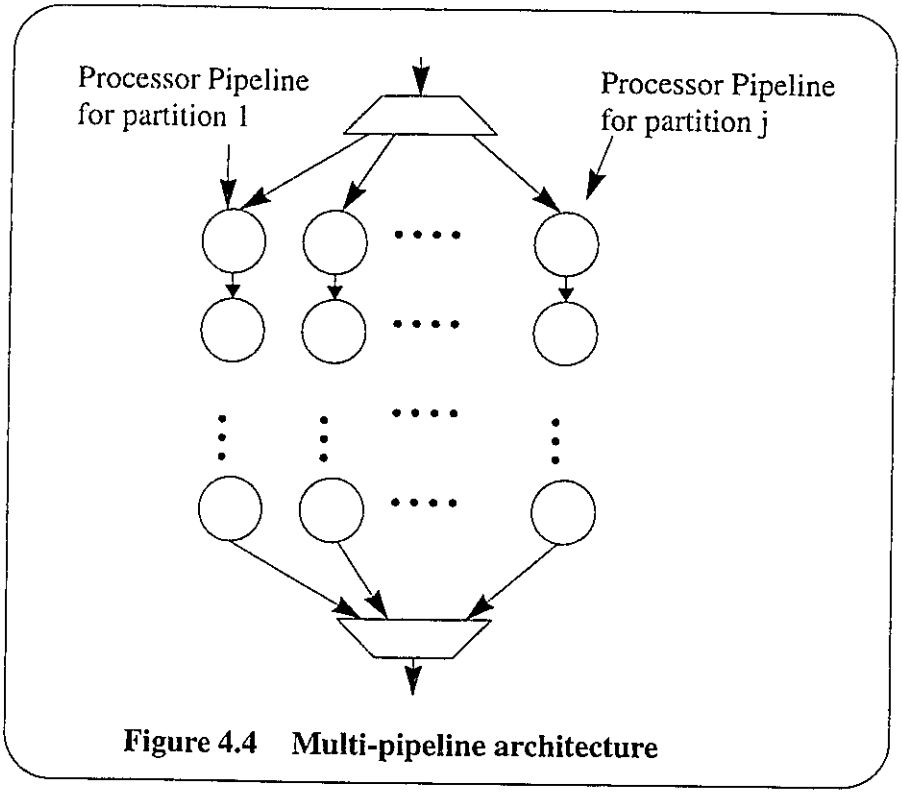


Figure 4.4 Multi-pipeline architecture

are left idle at different times (based on user partition mix ratio or workload characteristics). El-Amaway *et al.* propose a scheme which uses a series of pipeline reservation tables to be determined using an algorithm which is provably correct [91].

Shin *et al.* propose a scheme for reserving a fraction of execution time on a processor to guarantee execution of periodic tasks and maximize the probability of execution of aperiodic tasks [90]. Smirni *et al.* show that processors should be added to maximize the residual capacity of the array [88]. That is, the processors should be added so as to provide a 'buffer' of processing capacity which can be used as a reserve for unpredictable changes in the workload intensity or sub-call mix. Idle processors can be justified when:

- applications do not have good speedup characteristics (Call Processing speedup can be limited by the need for access to large, centralized databases).
- when the system utilization is medium to high (this is true for most Call Processing systems).

- when the job arrival rate is highly variable (this is also true for most Call Processing systems).
- when the system size is large (not the case for most current Call Processing cores.)
- when the workload is multi-class (this is the trend as the number of call types increase; however, only one call type has been simulated here).

Since most of these are characteristics of the Call Processing problem, idle processors may be useful in improving scaleability. The throughput of the two architectures, for one to ten processors, has been graphed in Fig. 4.5. The Linear array throughput advances in steps at processor counts where the bottleneck assignments change because the load can be redistributed evenly again. Throughput can remain constant as a processor is added because of a processing bottleneck within the array. The Linear Array does not scale as smoothly as the Load sharing architecture, whose throughput in the ideal case scales linearly as new processors are added.

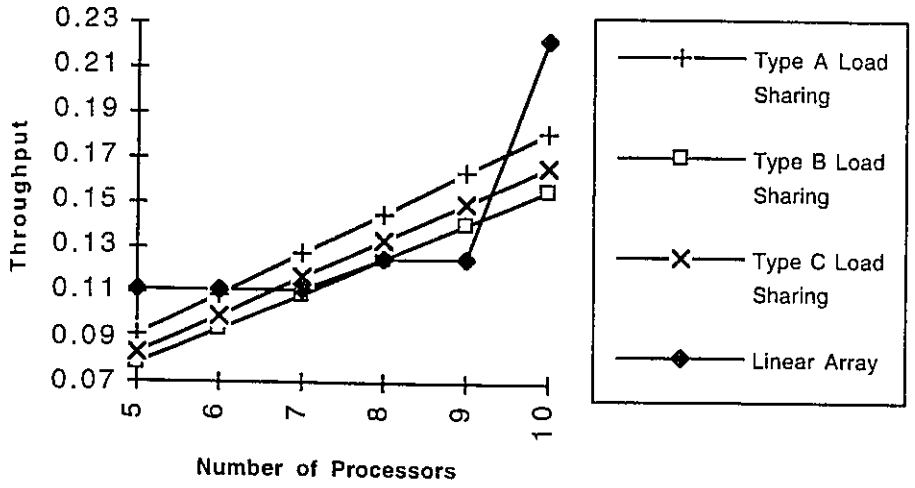


Figure 4.5 Throughput and Cost Factor Scaling for Load Sharing and Linear Array

Even if processors are added, performance will not improve, unless the delay associated with the bottleneck operation can be reduced. This can be done using a more specialized processor or by

dividing the task into smaller sub-tasks. The second approach is less applicable to Call Processing applications, since the largest delays are associated with the access to centralized data (e.g. routing, customer billing and service/call authorization; see Appendix A for more details). New services and new service interactions complicate the situation further and can increase the number of accesses to the busy central databases.

Distributed databases are an alternative and could allow more efficient access by multiple processors. However, the user databases remain centralized for several reasons:

- Telecommunication switch architectures have typically been based around a single processor and re-designing customer databases is time-consuming and costly.
- Newer services can be initiated on a per-call basis and require more interaction with each other. As this trend continues, or develops in unanticipated ways, keeping distributed copies of the database consistent gets harder.
- Geographical and network boundaries as well as economies of scale all act to further centralize the storage of customer data.
- Telco service providers **want** to maintain tight control over the database for security and economic reasons.

The database centralization constraint limits the scaleability of multi-processor architectures for Call Processing applications. Thus, database access contributes heavily to Amdahl's fraction (which is the fraction of work that must be run sequentially [19]). Therefore, to increase performance of the Call Processing architectures, the biggest improvements are achieved by reducing the processing times associated with the time-consuming access to the central databases mentioned above.

In the linear array, a specialized processor or group of processors can be assigned to the database access function. In the homogenous Load Sharing architecture, all processing elements compete equally for access. The resulting demand is 'bursty' and is not automatically regulated as it is

with the linear array. The effect on the Load Sharing architecture is to increase the delays experienced by users.

The Linear Pipelined Array looks very promising. It allows specialization and implements the Call Processing application efficiently, because of the inherent bottlenecks in the application described earlier. The interconnection topology is also very efficient and suitable for the majority of calls which have to be processed. More detailed work is now required to determine the limits on performance of the Linear Array for a Call Processing application. Processor specialization is required for this architecture to perform well [92]. The remainder of this work assumes that processor specializations (or small 'processor clusters') are used to handle any processor-intensive operations.

4.2 Summary

In this chapter, a new architecture for Call Processing has been described - the Linear Pipelined Array. Using the design criteria generated in Section 2, each architecture described in Section 3 has been evaluated and compared. The throughput of the Linear Pipelined Array and the Load Sharing architecture have been compared in detail. Processor specialization has been shown to be justified, which results in a heterogenous architecture. A cost factor has been introduced, for normalization purposes, which allows fairer comparison between different heterogenous architectures.

The Linear array does not scale as well as the Load Sharing architecture, but adding extra processor delay stages to the array or using multiple arrays can both increase throughput. Since the need for a centralized database limits the scaleability of the Call Processing application, itself, the Linear Pipelined Array is a promising approach for a Call Processing implementation and is worthy of more detailed investigation. This investigation is the subject of subsequent chapters.

Chapter 5

Call Processing Simulation for the Linear Pipelined Array

This chapter describes the simulations performed to evaluate the suitability of a linear pipelined array for a Call Processing implementation. The structure of the model is given, followed by the input and output parameters which are varied and observed, respectively. For this work, the key performance criteria are [16]:

- latency (time to process a single call).
- throughput (number of calls processed per second).
- response time (e.g. dial tone delay, digit collection, routing and connection) to users.

5.1 Processor Assignment and Scheduling

To evaluate the linear array, it is necessary to study the assignment of program modules to processors and the scheduling of these modules for execution to guarantee that deadlines are met. Many of the scheduling issues associated with this problem are discussed in [68] and a comprehensive taxonomy of scheduling methods is presented in [93]. Stankovic *et al.* have examined some of the constraints imposed when developing schedules to meet hard 'real-time' deadlines [70].

For a pipelined linear array, load balancing amongst the stages is important [82]. The maximum throughput is limited by the delay of the slowest stage in the array. There is some research that shows that the judicious introduction of extra delay can make workloads smoother and increase pipeline throughput [89]. Extra processors can be used for multi-processor load-balancing techniques [88]. The scheduling decision of which processor to leave idle can be dynamic, where different processors are left idle at different times (based on call arrival workload characteristics).

An alternative to leaving processors idle is to create extra pipelines which use them more efficiently [85, 86, 87].

Bottlenecks in performance mean that very different assignments will perform at the same level if they share a bottleneck assignment. Sigalov *et al.* state that a dynamic programming approach is required to find a solution for more than three processors [94].

For the case where the number of processors is less than the number of modules, Iqbal *et al.* have proposed an algorithm of polynomial complexity for scheduling chain-like tasks onto a linear array of homogenous processors [79]. This has been further speeded up by Nicol *et al.* [95]. Stone applied Network flow algorithms, originally described by Ford *et al.* [97], to derive an optimal polynomial solution to the general two-processor assignment problem [96]. Finally, Lee *et al.* have extended Stone's approach to derive the optimal static assignment of a distributed program of any structure onto a linear array [73]; this is useful for future Call Processing services with unanticipated precedence relationships and requirements [98, 99].

As with most scheduling-related problems the following questions are important [68]:

- Can the problem be solved in polynomial time (or is it NP-hard or NP-complete)?
- If the problem is in the NP class, is there a heuristic approach which yields adequate or near-optimal performance for the application (e.g. Call Processing)?
- If a heuristic approach is adopted, can it be shown to be close enough to optimal for all or for a large enough set of input loading conditions and system states?
- Does the scheduling heuristic scale well with different numbers of processors? For the Call Processing application, this corresponds to whether switches of different sizes can be built and/or how many users a given size of architecture can support for a variety of realistic loading conditions.

5.2 Model Description (Verilog Code Structure and Organization)

Two simulation methods have been used in previous work for the modelling of a multi-processor system. The first and most common is the event-driven approach [100, 101, 102, 103], in which the events are typically different stages of program execution. The second approach is to model the execution of the program directly, by representing the hardware as constraints placed upon program execution [104]. The results of the trace-driven approach are then directly comparable with trace data taken from real program execution. In this work, the event-driven approach has been selected because of its flexibility and because the trace-driven approach involves the use of data which is specific to a single telecommunication switching system and extrapolation from this data is difficult.

Verilog [105] is an event-driven [106] simulation language used to describe ASIC, PCB or system-level hardware at a behavioural level. Verilog has stochastic capabilities which allow random events to be generated with specific statistical properties, as well as queue management and queue statistics-gathering functions. These are useful for the stochastic modelling of call arrivals and user behaviour and for the pipeline array model, itself, where events are queued at the input to each processor. Verilog has been used for the simulations presented here.

The basic Verilog structural entity is a module, which has a defined interface and content and can be instantiated (instances created of the module) multiple times, hierarchically if required. Each module supports procedural tasks and functions which exist only within the context of that module. Two procedures with the same name cannot be executed simultaneously within a module. Rather, subsequent calls to a procedure supercede previous calls, over-writing internal variables in the task.

In this simulation, modules are used whenever there is an identifiable hardware element to be modelled. Tasks and functions are used for distinct procedures within a module. The simulation token is a single call, which is labelled with a Call/Job ID number. This number is used as an index to call-specific data structures (e.g. the call state variable).

The simulation modules (a representative 3-processor system is shown in Fig. 5.1) are:

- *top*: *top* is a single top-level module which contains instantiations of all of the processor modules and all of the tasks associated with the generation of traffic, the collection of traffic statistics and the production of data out of the simulation. This is accomplished as follows:

  - *generate*: Simulation tokens are generated and labelled with an ID number. These tokens are scheduled to appear at the input of the first processor in the array at random times. Note that simultaneous calls are separated by an artificial delay of 1 time unit, which is an acceptable limitation since the calls are queued at the first processor anyway before being processed.

  - *statistics*: Simulation tokens emerging from the last processor are examined to determine if processing has been completed or not. If not, the token is re-scheduled to appear at the input of the first processor at a random time. If processing is complete, various data are logged for the call and the ID number is recycled back into a queue of IDs. Once 1000 calls have been processed, the processors are flushed clean of jobs ready for the next simulation run, if required.

  - *output_results*: Prior to job flushing, the results for the run are displayed as printed output (which is post-processed later into a graphical form).

- *proc*: *proc*: is a module containing the queues, buses and logical elements used to model a processor. This module instantiates a single version of the scheduler module (see below). The inputs accepted into *proc* are:

  - job *ID*s transmitted from the 'upstream' processor,

  - a control signal (*sig*) to indicate when a job ID is valid,

  - a *ready* signal to indicate that a downstream processor is capable of accepting output from this processor module. A simple 'stop and wait' protocol is used for flow control.

The outputs produced are:

- job *ID*s of the processed calls (driven from the *scheduler*),

- a control signal (*sig*) to indicate when a job ID is valid,

- a *ready* signal to indicate to an 'upstream' processor that this processor is capable of accepting input.

- *scheduler*: This is a module containing a logical task (*process_or_not*) which decides whether to process (or pass through) the sub-calls/program modules of each call and when to remove input data from the queue. The scheduler also contains a *CPU* task which implements an atomic delay operation to simulate the execution of instructions. The inputs accepted into the *scheduler* module are:

  - job *ID*s received on request from an input queue in the *proc* module,

  - a control signal (*rx_q_empty*) to indicate whether this queue is empty or not.

  The outputs produced are:

  - job *ID*s of processed items,

  - a control signal (*job_out_flag*) which indicates job ID validity, once the processing time for the sub-call/program module has elapsed.

The complete Verilog implementation of the simulation is given in Appendix A. To illustrate how the simulation works, the processing of a single Call/Job ID is described in the following paragraphs (refer to Fig. 5.1):

(1) A Call is represented by a unique job ID, x, and is generated in *generate.v* using a particular random generation function (e.g. exponential [53]).

(2) The Call is held until the queue in the first processor of the array (processor 0) is able to accept the call. Other calls can be generated during this holding time. The Lost Calls Cleared (LCC) assumption* has been used here. So, if processor 0 is not ready, the token

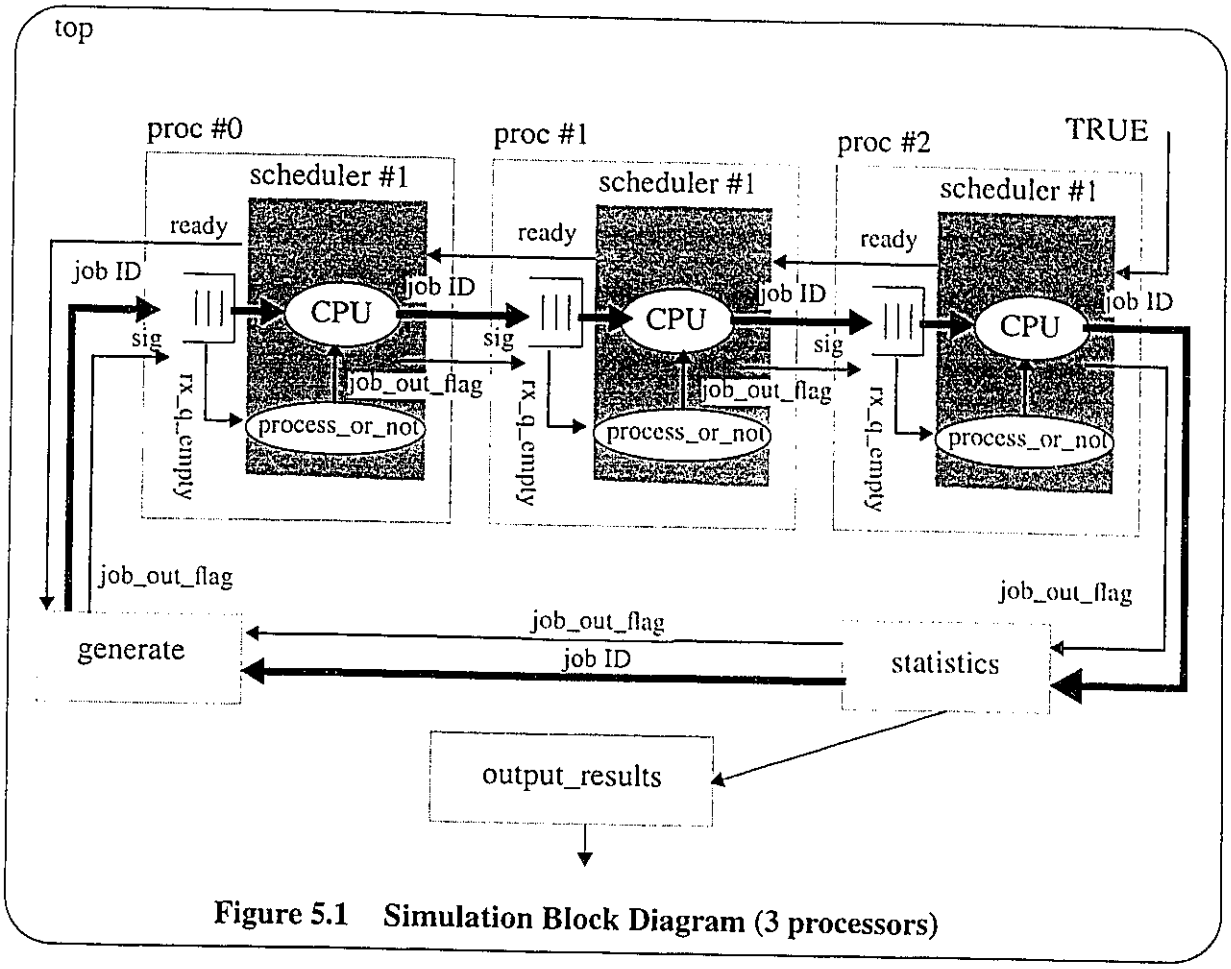


Figure 5.1 Simulation Block Diagram (3 processors)

is destroyed. If processor 0 is ready, the job ID is queued there. The process state variable is initialized to the starting state value and the following variables are reset appropriately:

(A) birth time,

---

* The 'Lost Calls-Cleared' (LCC) model assumes that users give up and try again later (when the switch is less busy) if they are not served. This is a reasonable assumption for real traffic [55] and is used here. Calls which find the system too busy disappear with zero holding time. The re-trials do not, therefore, change the pattern of call arrivals. This traffic modelling assumption is used in all countries, besides North America, which uses the 'Lost Calls-Held' (LCH) assumption [55].

In the LCH model, users who do not get dial tone soon enough or who have misdialled, are often assumed to re-dial immediately, rather than waiting for a large time before re-trying. Thus, a busy switch gets busier as more time is wasted handling calls which are going to be abandoned and attempted again. There is disagreement as to the validity of this model [107].

(B) total delay time associated with waiting for user input and responses,

(C) total processing time.

(3) Once the scheduler in processor 0 detects the new job, it removes job x from the first-come first-served FIFO queue for processing. The ID number, x, is used as the index to look up the job state (currently at the initial value). The *process_or_not* task decides whether to process the call or not. This decision is made after considering such factors as (see Chapter 6):

(A) the local queue size,

(B) the call state probabilities (relative ratios of calls in particular states; this can be determined from local or global information),

(C) the suitability of the processor for processing the particular call state.

(4) If processing is required, the CPU task blocks itself active for the appropriate amount of time, before updating the job state variable for the call.

(5) If no processing is required, the job state variable is not altered and there is no delay.

(6) The job is sent into the queue of the next processor in the array.

(7) The architecture is engineered not to drop calls once they have been accepted by the first processor. Therefore, the case where subsequent processors are unable to accept the call is avoided by making the queues in the *proc* modules large enough. Process steps (3) to (6) are repeated until processing is completed, a user break is encountered, or the end of the processor array is reached.

(8) If processing is not completed or a user break is encountered, the code in *statistics.v* schedules the job to wake up again after a delay determined by a particular random number generation function (e.g. exponential), which is call state-dependent.

(9) If processing has been completed, the job ID is returned to the 'pool' of job IDs and

various data is recorded for the call such as:

(A) total processing time,

(B) total delay associated with waiting for a user response or input,

(C) total system overhead associated with the processing of the call (calculated as the total duration of the call minus the user and processing delays).

(10) At the scheduled time, the job ID which requires further processing 'wakes up' with the old job state and is scheduled for processing on the array again. This 'old' job competes with the 'new' jobs for processing time on the array, which proceeds as before (step (2) and on).

5.3 Call Characteristics

The simulation uses the originating call model (Appendix A.1), which has ten sub-call states. These sub-calls may be combined into a smaller number of sub-calls for ease of implementation, but this has not been done here. The amount of time required to execute each individual sub-call for different processor types depends on the system state and call mixes as well as on the basic hardware and software capabilities.

Reasonable estimates of sub-call work times are made using an average of the call work times given in [108]. The DMS-100 system uses the Motorola 68000 series of processors and [108] gives work times for a variety of call mixes. Average values can be calculated for a mid-range processor and a typical call mix and these are given in Table 5.1, yielding an average work time of less than half a millisecond per call on any of the five processors.

To model the processor specialization, which is desired for the simulation, adjustments up and down around these means have been made for each of the ten sub-call states, to indicate that specific processors are more suitable for particular states. The degree of variation used depends on the variation which was observed when the processor type or the call mix was varied. The values have been rounded to the nearest 10 μs.

Module / *Processor*	0	1	2	3	4	5	6	7	8	9	Total (μs)
0	100	30	30	60	20	40	20	50	30	40	420
1	90	50	40	50	50	30	20	40	20	70	461
2	100	70	30	30	30	40	30	50	40	50	470
3	20	90	40	60	20	50	50	50	40	20	440
4	30	30	50	50	30	20	20	50	40	90	410
Maximum Ideal Throughput (no communications) = (1/maximum processor delay) = $1/(40\times10^{-6}+90\times10^{-6})$ calls/s											

Table 5.1 Standard Processor/Module Execution Time Array (in μs)

Various internal processor parameters have also been approximated. These are:

- Compress and Decompress delay: Calls are assumed to be represented as a data structure which is stored in memory in a compact form. The condensed data structure needs to be unpacked prior to the execution of the constituent sub-calls. A value of 5 μs is used for the simulation runs*.

- Transmit and Receive messaging delay: The Linear array processors communicate using a messaging mechanism. These delays are the time required to send and receive a message, respectively. A value of 5 μs is used for the messaging delays.

- Retrieve delay: This is the delay associated with de-queuing a message from the queue of received messages. This delay is independent of the receive message delay and models the time to move data into working memory. A value of 5 μs has been assumed for this parameter.

---

* 5 μs is the time taken to execute 100 memory accesses with 50 ns (current DRAM access cycle time) required for each access. If more memory accesses or additional processor operations are required, memory cycle times need to be reduced (faster memory technology or interleaving techniques). Similarly, if fewer accesses/operations are required, slower memory technology can be used. This modelling assumption has been made for the estimation of other delays which are highly dependent on the exact implementation requirements.

- Interrupt Service Routine Passthrough delay: An efficient mechanism to detect that the end of processing has been reached is assumed (i.e. this passthrough mechanism could exist in the Interrupt Service routine of the processor [83]). A value of 5 μs is assumed for this delay.

- Scheduler passthrough: If the end of processing has not been reached, the scheduler must determine whether processing is appropriate or not for the particular sub-call. If it is not, a penalty delay of 5 μs is incurred, which is the delay associated with detecting that processing has been completed.

- Scheduler execution time: To simplify comparison, the scheduler is assumed to execute in zero time regardless of the complexity of method used. This is an acceptable assumption, since efficient scheduling algorithms have been implemented on a multi-processor pipeline in [109] based on the network flow algorithms originally proposed in [97].

5.4 Scheduling Strategies

There are several methods for the assignment of sub-call tasks to processors [68, 70, 93]. Scheduling can be either static or dynamic. Most classical scheduling theory deals with static scheduling. Here, a complete knowledge of the set of tasks to be executed and the constraints on execution are known (e.g. computation times, precedence constraints, deadlines). Dynamic scheduling has the same complete knowledge of sub-call tasks, but new tasks can arrive, which were unknown when the schedule was derived. Therefore, the dynamic schedule changes over time and cannot be pre-computed [70].

5.4.1 Basic Theory

The simplest possible scheduling strategy is to try all possible assignments of modules to processors and then choose the assignment that gives the greatest throughput. Generating all possible assignments of program modules/ sub-calls to processors can be achieved in $\Theta(nm \log n)$ operations, where m is the number of modules and n is the number of processors [79]. This has

been further improved, using a dynamic programming algorithm, to $\Theta(nm)$ by Olstad *et al.* [110]. The problem of generating all possible assignments of m=10 distinct modules to n=5 processors is identical to the problem of generating all possible distinct arrangements of five addends which sum exactly to ten. For example, the set {1,2,2,1,4} means:

- the first module (module 1) is assigned to processor 1,
- modules 2 and 3 are assigned to processor 2,
- modules 4 and 5 are assigned to processor 3,
- module 6 is assigned to processor 4 and
- the last four modules (modules 7, 8, 9 and 10) are assigned to processor 5.

Zero values, if permitted, correspond to processors which are not assigned any work. The number of arrangements of n objects, of which there are x_i things of the i^{th} kind is given by the multinomial coefficient [111]:

$$\begin{pmatrix} n \\ x_1\ x_2\ x_3\ \ldots\ \ldots\ \ldots\ x_k \end{pmatrix} = \frac{n!}{x_1! \cdot x_2! \cdot x_3! \cdot \ldots\ldots \cdot x_k!}$$

For Table 5.2, the multinomial coefficients are:

$$\begin{pmatrix} 5 \\ 2\ \ 2\ \ 1 \end{pmatrix} = \frac{5 \times 4 \times 3}{2} = 30$$

$$\begin{pmatrix} 5 \\ 3\ \ 1\ \ 1 \end{pmatrix} = \frac{5 \times 4}{1} = 20$$

$$\begin{pmatrix} 5 \\ 4\ \ 1 \end{pmatrix} = 5$$

$$\begin{pmatrix} 5 \\ 5 \end{pmatrix} = 1$$

If processor 'skipping' is not allowed, then Table 5.2 gives all of the 126 possible arrangements of the 7 possible feasible sets of five non-zero addends which add up to 10 (m=10, n=5 case).

Addends / *Index counter*	*2,2,2,2,2*	*1,2,2,2,3*	*1,1,2,3,3*	*1,1,2,2,4*	*1,1,1,3,4*	*1,1,1,2,5*	*1,1,1,1,6*
1	22222	23122	23113	24112	14113	15112	16111
2		22132	32131	22141	31141	21151	11161
3		21322	31321	21421	31411	21511	11611
4		21232	31231	21241	14131	15121	61111
5		22321	32311	22411	41311	51211	11116
6		23212	23131	24121	11431	11521	
7		23221	31312	41212	43111	52111	
8		32212	12331	12421	41113	51112	
9		32122	33112	42112	13411	12511	
10		12322	31123	41122	41131	51121	
11		32221	13312	12412	11134	11125	
12		31222	31132	41221	11143	11152	
13		12223	11332	11422	11314	11215	
14		12232	33211	42211	13114	12115	
15		22213	32113	11224	13141	12151	
16		21223	11233	11242	31114	21115	
17		13222	12313	12214	34111	25111	
18		22312	12133	12142	11413	11512	
19		22231	13123	12124	14311	15211	
20		22123	13132	21214	11341	11251	
21			31213	21142			
22			13231	14122			
23			33121	12241			
24			13213	21124			
25			21133	14221			
26			11323	22114			
27			13321	14212			
28			23311	42121			
29			21331	21412			
30			21313	24211			

Table 5.2 Possible Assignments (without skipping) for ten modules on five processors

If processor 'skipping' is allowed, the number of ways a sum to m=10 can be constructed in n=5 digits increases from 7 to 30 (zero digits allowed) and the number of distinct arrangements increases from 126 to 1001. Table 5.3 shows the 30 possible sets of 5-digit addends that sum to 10. Shaded entries in the table contain no zeroes and their arrangements have already been listed

fully in Table 5.2.

Digits	*Number of Arrangements*
0000a	5
00055	10
00046	20
00037	20
00334	30
00028	20
00244	30
00235	60
00226	30
02233	30
02224	20
22222	1
00019	20
00145	60
00136	60

Digits	*Number of Arrangements*
01333	20
00127	60
01234	120
01225	60
12223	20
00118	30
01144	30
01135	60
01126	60
11233	30
11224	30
01117	20
11134	20
11125	20
11116	5

Table 5.3 Numbers of Arrangements for the assignments of 10 modules to five processors

A static assignment strategy based on only local information has the advantage that it does not require large amounts of communications between the processors or between processors and a centralized scheduler, but it does not adapt well to new call arrival patterns. Dynamic re-arrangement strategies are adaptive [85, 86], but increase the complexity of the system significantly.

5.4.2 Scheduling based on Call Mix

The optimal schedule for the pipelined array (as already explained) is the schedule which minimizes the maximum sub-call execution time over all processors, as the job mix varies. For changing switch services or changing service mixes, it is difficult to fix an optimal order of the linear pipelined array, since the best schedule at a given time depends on the 'mix' of sub-calls. Strategies which choose the best order for the average job mix must be used instead. Thus, a job

mix ratio is established and is used to weight the delays for specific modules being executed on particular processors.

Mathematically, let α_{ij} be the entries in an execution time matrix, where each entry gives the execution time of module j on processor i. Then, $\overline{\alpha}$ is given as follows:

$$\overline{\alpha} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1m} \\ \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ \alpha_{n1} & \alpha_{n2} & \cdots & \alpha_{nm} \end{bmatrix}$$

Let γ_j be the relative frequency (analogous to the probability over a large time period) of the sub-call and note that the sub-call belongs to a particular partition, δ_k, where there are j partitions. A partition is a set of sub-calls which can be scheduled atomically. For example, a convenient partition is at a break for user input. Then, the entries $\overline{\alpha}$ can be weighted by the probability of the partition, which can be estimated by γ_j. It is possible to derive an 'optimal' schedule based on the recent job history using the new weighted execution time entries, $\overline{\alpha}$', where $\overline{\alpha}$' is given as:

$$\overline{\alpha}' = \begin{bmatrix} \gamma_1\alpha_{11} & \gamma_2\alpha_{12} & \cdots & \gamma_j\alpha_{1m} \\ \gamma_1\alpha_{21} & \gamma_2\alpha_{22} & \cdots & \gamma_j\alpha_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ \gamma_1\alpha_{n1} & \gamma_2\alpha_{n2} & \cdots & \gamma_j\alpha_{nm} \end{bmatrix}; \text{ with } \sum_{i=1}^{j} \gamma_i = 1$$

This strategy does not react well during periods of extreme or intense change in the job mix, but it serves as an interesting benchmark. Two alternatives are simulated, using pre-calculated schedules determined for combinations of partitions, derived from the weighted execution time matrix, $\overline{\alpha}$':

- Locally derived counts of partition occurrence are used to 'look up' the appropriate optimal schedule. That is, a queue of the most recent 100 partitions is maintained at each processor and the 'nearest' pre-calculated ratio to the measured ratio is used. The corresponding pre-calculated optimal schedule for it is then applied.

- The head-end processor alone determines the schedule. This processor communicates its schedule down the array and all other processors adopt the selected assignment. This method has the advantage of being the most responsive to changes in job mix (no delays for data to propagate from upstream processors) as well as not requiring sophisticated co-ordination amongst the processors to ensure that schedules in each of the processors are consistent.

5.4.3 Heuristic Rules for Scheduling

Deriving the criteria for optimal schedules (such as minimize the maximum execution time) is analogous to a searching problem in the NP-hard domain. The general assignment problem for more than two processors in a linear array is shown to be NP-hard in [112]. Although an exhaustive search of solution spaces was used in the simulation, these 'brute force' techniques are not feasible for large numbers of program modules and/or processors. Various other searching techniques have been proposed in the literature for searching on a linear array [83, 113] and it may be possible to adapt these to decrease the searching time.

Often, however, heuristic rules are used to decrease the size of the large problem search space. A good heuristic prunes the complete search space down to a manageable size without discarding too many high-quality solutions and does so in a reasonable amount of time [114]. The result may not be absolutely the best but should be 'close enough'. To evaluate a heuristic, the following approaches can be used:

- relative comparisons of different heuristic rules are made (this is the main approach taken in this work).

- relative evaluations of heuristic rules in comparison with the optimal solution are used (this approach is used for the smaller examples in this work).
- a wide variety of input conditions are used to establish the limits of performance for the heuristic. Note that few heuristics handle a wide variety of problem spaces gracefully and a range of input conditions are used for the examples in this work.

The heuristic rules used in the simulations are:

- Process if better than average at processing the particular module, otherwise default to another possible assignment.
- Process if better than average, otherwise pass-through if the worst processor for a given module (or use a default assignment failing that).
- Process if better than average, otherwise pass-through if the worst or next-to-worst.
- Process if the best and pass-through if the worst.

5.5 Ordering of Processors in the Linear Array

Since each call flows through the processors in order from the first processor to the last, the first processors usually perform the first sub-calls and the later processors perform the later sub-calls. The placement of each processor in the array, therefore, affects the performance of the array. By varying the order of the processors, it is possible to generate more productive assignments. For example, a more optimal ordering of the same five processors discussed in Section 5.3 is given in Table 5.1 (the same processor labels are used in both tables).

In both cases, the shaded portion of the table gives the critical path for execution. Simulation results for both of these orderings are given in Chapter 6. Note that, for a large variety of call types, it is difficult to determine a time-invariant universally optimal processor ordering. Therefore, a 'close-to-optimal' processor ordering has been used in the simulations.

Module / *Processor*	0	1	2	3	4	5	6	7	8	9	Total (μs)
4	30	30	50	50	30	20	20	50	40	90	410
2	100	70	30	30	30	40	30	50	40	50	470
0	100	30	30	60	20	40	20	50	30	40	420
1	90	50	40	50	50	30	20	40	20	70	461
3	20	90	40	60	20	50	50	50	40	20	440
Maximum Ideal Throughput = (1/maximum processor delay) = $1/60\times10^{-6}$ calls/s											

Table 5.4 Optimally ordered Standard Processor/Module Execution Time Array (in μs)

5.6 Network Loading Conditions

Telecommunication switches are engineered for high-day busy-hour traffic (HDBH) conditions [51]. That is, a switch is deployed with enough capacity to handle the required number of lines with the required performance during the busiest calling period. Excess load is rejected during these periods in a controlled fashion using an overload-handling mechanism [107, 115, 116, 117, 118]. During these busy periods, the queue of jobs waiting to be processed is close to its maximum size and new calls are rejected (no dial tone supplied to the user).

The switching network traffic changes with the season, month, week, day and hour [51], hence the Call Processing architecture must be able to handle different loading conditions and different types of loading statistics. The two basic load types are:

(1) uniform distribution for delays between calls, which would occur if the input call rate was being limited into the architecture. Loading rates are varied to simulate different calling periods. The architecture starts to overload when the interval between jobs is about 500 μs, using the processor delays and call timings given in the previous chapter. Since most telephone switches are engineered for the heaviest load condition, the majority of the simulation work and performance comparison work has been performed at higher offered loading levels than one call every 500 μs. In this way, more significant comparisons can be made, since a faster architecture will not starve for work. Intervals of 500 μs, 300 μs,

100 μs, 50 μs and 25 μs have been used.

(2) exponential distribution for delays between calls is used, which assumes independently generated calls and no throttling mechanism. Independent call arrivals is a valid assumption for a large telephone network [54]. Simulations are performed with exponential distributions having means of 100 μs, 50 μs and 25 μs.

5.7 User Delays

As telephone services change, the nature of the user interaction can vary. The effect of adding user delays into the processing of calls is simulated and compared with the contiguous and uninterrupted processing case.

At the second and sixth state in the ten-state originating call state machine, there are potentially large delays associated with the authorization of the origination attempt and the authorization of call setup, since these require access to (possibly external) databases. Therefore, calls are passed through and out of the array once these states are reached. These user delays are modelled using an exponentially distributed random variable with a mean of 50 μs; individual user delays will be on the order of seconds. As stated earlier, a peripheral/core architecture has been assumed and this delay is the assumed response back from the peripheral to the core. Because of the memoryless property of the exponential distribution, this is a reasonable simplification (which reduces array sizes in the simulation).

5.8 Scaleability of the Linear Array

Simulations with 5 and 8 processors were performed. The processor ordering shown in Table 5.1 was used for the 5-processor architecture. To derive an 8-processor architecture, an extra three processors were added to the array at the end, which were identical to the first three processors in the original 5-processor array. That, is the 8-processor ordering is: 0,1,2,3,4,0,1,2, with a 5-processor ordering of: 0,1,2,3,4. Adding more than n=10 processors introduces idle processors and has not been simulated here.

5.9 Output Parameters

The following parameters derived from the simulations are examined for each of the trials:

- latency of a single call (mean and variance),
- mean CPU utilizations of all processors in the array for 1000 processed calls,
- standard deviation of the CPU utilizations of all processors in the array for 1000 processed calls,
- mean CPU utilizations for each processor for 1000 processed calls,
- overall throughput rate for the 1000 processed calls,
- response times for individual calls,
- number of rejected calls.

The standard deviation of the mean throughput is calculated in the simulations as [119]:

$$s = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n-1}}$$

$$= \sqrt{\frac{\sum (x_i^2) - n \bullet \bar{x}^2}{(n-1)}}$$

$$= \sqrt{\frac{\sum x_i^2}{(n-1)} - \frac{(\sum x_i)^2}{n \bullet (n-1)}}$$

where:

- x_i = total elapsed CPU utilization time for the i^{th} call,
- n = 1000 = total number of calls

The mean and the standard deviation of the system throughput give a good statistical description of the efficiency and stability of the system. By also examining the utilization of each processor, the performance bottleneck trends can be observed. A good indication of switch service 'quality'

can also be obtained by examining the response times for individual calls for representative simulation cases and by looking at the statistics for rejected calls.

5.10 Summary

In this chapter, the simulation of the Linear Pipelined array for a Call Processing application has been described fully. The modelling assumptions made and the detailed organization of the simulation have been described. The input variables to be adjusted and the output parameters to be measured have also been described.

A theoretical discussion and analysis of the Linear Pipelined array performance has been given for the following areas:

- Scheduling approaches.
- Network loading effects.
- Processor ordering effects.
- User Delays and their effect
- Scaleability issues.

The expectations for the simulation results have been stated. This theory provides the simulation plan, whose results are presented in Section 6.

Chapter 6

Performance Analysis

This chapter presents the results of the simulation experiments. In each subsection, the simulations are described, the results are presented in graphs or in a table and the data is analyzed and interpreted. A clear description of experimental organization and accurate presentation of results can be found in [120] and many of the suggestions in this reference have been used in this thesis. The simulations are organized into the following categories:

- Effects of Assignment/scheduling strategies of sub-calls onto processors,
- Effects of presence or absence of a delay for user input,
- Effects of changes in the input traffic conditions: changes in the statistical distribution and/or frequency of new call attempts,
- Effects of increasing the number of processors.

6.1 Assignment/ Scheduling Strategies

The standard 5-processor architecture to process 10 modules has been simulated with different assignment schemes. In each of the simulations, a total of 1000 calls were processed for each test case. This is adequate to stabilize the architecture so that the CPU utilization means no longer change significantly. For all of these different schemes, an input traffic rate of one new call every 50 μs has been used. This ensures that the performance is limited by the architecture rather than by the supply of new call requests. User delays have been assumed (Section 5.7) to keep the simulations representative of the Call Processing application.

6.1.1 Scheme Descriptions

The following assignment schemes have been examined:

(1) All 126 possible assignments (listed in Section 5.4.1) which do not permit processor skipping, but which could complete processing of all of the 10 sub-calls/modules in a single pass through the array (termed 'non-skipping assignments') are simulated*.

(2) If a processor is 'better than average' at processing a given module, the module is processed, otherwise one of the 126 possible non-skipping assignments described above is used. All of these assignment cases are simulated (heuristic 1). 'Better than Average' means that the processor has an execution time for a given module which is lower than the mean execution time for that module, taken over all processors in the array.

(3) If the processor is the fastest processor for a given module, the module is processed. If the processor is the slowest processor for a given module, the module is passed through without processing. If neither of these cases are true, one of the 126 possible non-skipping assignments is used. All of these assignment cases are simulated (heuristic 2).

(4) All 1001§ possible assignments (listed in Section 5.4.1) which can complete processing of all of the 10 modules in a single pass through the array (termed 'skipping assignments') are simulated. Note that any of the processors are allowed to be skipped for these assignments.

(5) If the processor is better than average at processing a given module, the module is processed. Otherwise, one of the 1001 possible skipping assignments is used. All of these assignment cases are simulated (heuristic 1A).

---

* Calls are not typically processed in a single pass, because of user delays which are encountered between modules (see Section 5.7).

§ Because of the single-pass restriction on generated schedules, these do not represent all of the possible assignments of ten modules to 5 processors; there are 5^{10}=9,765,625 of these and they have not all been simulated because of inadequate simulation resources and time constraints.

(6) If the queue size exceeds a threshold value, modules are passed through without processing, otherwise processing is performed according to one of the 126 possible non-skipping assignments. All of these assignment cases are simulated (Fig. 6.11).

(7) The call mix ratio is estimated using the relative number of calls in each of the possible user partitions at any time during the simulation and is used to select the fastest assignment (see Section 5.4.2). Here, processors are allowed to be skipped, with the restriction that each user partition must be processed in a single pass through the array. Two call mix ratio estimation strategies are used, but both are based on counting the number of calls currently in each user partition (see Section 5.4.2). They differ in the way that the calls in each partition are counted; for both schemes, 126 runs of 1000 calls are simulated:

(A) Counts are maintained in each processor and used only in that processor (Fig. 6.12). This means that different processors can have differing views of the current call mix during the simulation and can select different assignments from each other. However, in general, the assignments will be the same and all modules will get processed.

(B) Counts are maintained in the headend processor and this single estimate is used by all other processors (Fig. 6.13). This means that all processors have identical views of the current call mix during the simulation and use the same assignment, changing these assignments synchronously. This avoids the problem of inconsistent assignments, but requires broadcasting of scheduling information.

6.1.2 Results

For assignment scheme (1), the mean and standard deviation of the throughput is shown in Fig. 6.1 along with the processor CPU utilizations. The results are ordered along the x axis in terms of increasing mean throughput and then in the order of decreasing variance for assignments with the same mean throughput. These results are also tabulated in Appendix C. In general, as processor utilization becomes more evenly balanced across the array, the mean throughput increases and the

variance of the mean throughput decreases (better and more predictable performance achieved if the load is balanced).

The overlay or cumulative sum of the individual CPU utilizations is also shown in Fig. 6.1, correlates very well with the mean throughput. Note that the total of all of the processor CPU utilizations in the array is shown as the upper boundary of the top darkly shaded portion. The throughput variance depends mostly on queueing delays in the array and increases if processors are congested. When congestion occurs, queues upstream of congested processors increase in size and the latency for each call increases.

The throughput means and throughput standard deviations for 1000 calls processed with assignment scheme (1) are shown in Fig. 6.2 and Fig. 6.3. A rejection threshold of 70 and 30 calls for the input queue size are used for these figures, respectively. Once the input queue size equals the threshold value, newly generated calls are rejected. The effect of decreasing the threshold point at which load is refused into any processor is shown by comparing these two figures.

As the threshold value is decreased from 150 to 30, the throughput variance decreases by an order of magnitude and the queue sizes at each processor also decrease. Because of the increased competition for queue space between new calls and recirculating old calls, both latency and queueing delay uncertainty increase. The mean throughput also increases marginally for all assignments; this is because processors seldom have to wait for work from backlogged upstream neighbours.

The bottlenecks in the architecture during a simulation run can be seen clearly by looking at the CPU utilizations. For the simulation run shown in Fig. 6.3, more detailed examinations of the CPU utilizations for the best and worst non-skipping assignment cases are given in Fig. 6.4 and Fig. 6.5, respectively. From the two detailed CPU utilization graphs, it can be seen that the throughput of processor 4 limits the overall mean throughput of the array for both the best and the worst cases.

As can be seen from Table 5.1, {1,2,3,2,2} is theoretically the best assignment (see Section 5.4 for an explanation of the notation) since it represents the partition with the smallest possible $\omega(\tau(p))$. However, in reality, the assignment {1,2,2,3,2} has a mean throughput of 2660 calls per second which is about 5% higher than the throughput of 2535 calls per second for the assignment {1,2,3,2,2}; see Fig. 6.5. This is due to processor 3 being used more efficiently, which prevents processor 4 from being overloaded as often.

Since processor 4 is the bottleneck processor in the array, as shown in Section 5.3, any overload on this processor causes more flow control back through the array and increased rejection of new calls at processor 0, thus decreasing throughput. This is shown in the graph of Fig. 6.4, which gives a more detailed view of cumulative CPU utilizations over all processors, combined with a display of the cumulative throughput rate for the entire array during the simulation run in Fig. 6.3 for {1,2,3,2,2} and {1,2,2,3,2}. The theoretical and actual best-case runs are marked (A) and (B), respectively.

However, although {1,2,2,3,2} gives better measured throughput, the theoretically best case assignment {1,2,3,2,2} results in processor CPU utilization percentages which are the closest to each other (i.e. have the lowest standard deviation) during a simulation run. This can be seen by examining Fig. 6.6, which shows the standard deviations of the 5 processor utilizations for each assignment vector, ordered in terms of increasing overall mean throughput. The theoretically best case assignment has a standard deviation for the processor CPU utilizations of 2.9%, while the measured best case standard deviation is 8.7%.

Theoretically, the worst case assignment is given by {6,1,1,1,1}, since this yields the highest possible bottleneck execution time and the largest possible value for $\omega(\tau(p))$. However, the assignment {5,1,1,1,2} has a mean throughput which is about 11% lower (1750 calls per second as opposed to 1935 calls per second; see Fig. 6.5). This is due to processor 4 gating the processed call output from the array. This causes rejection of new calls at processor 0 because of the flow control mechanism used.

The graph of Fig. 6.5 shows the theoretical {6,1,1,1,1} and actual {5,1,1,1,2} worst-case simulation runs of Fig. 6.3, which are marked as (A) and (B), respectively. Comparing the processor CPU utilizations in Fig. 6.5 with those of Fig. 6.4, the severe load imbalances which degrade throughput performance are clearly illustrated. Processor 0 is the initial bottleneck and is then, in turn, prevented from accepting new calls by downstream processors which are completing the processing of the calls accepted during the initial bottleneck period of processor 0. This cycle continues throughout the simulation, periodically, with the bottleneck point alternating between processor 0 and the remaining processors in the array. The performance for the very worst case is degraded by processor 4, which extends the period where the remaining processors in the array (other than processor 0) are the bottleneck. As before, the processor utilization standard deviations correlate well with theory and differ from each other by the largest amount (Fig. 6.5) for the theoretically worst case assignment given by {6,1,1,1,1}. The processor CPU utilization standard deviation for the theoretically worst case assignment {6,1,1,1,1} is 21.4% as opposed to the actual worst case assignment {5,1,1,1,2} which has a standard deviation of 18.6%.

The first heuristic (heuristic 1) which has been simulated is given by:

Heuristic 1

```
if ('better than average' at processing the given module) then
    process
else
    use one of the 126 assignments (where processors cannot be skipped)
    /* this is the default non-skipping static assignment */
```

Fig. 6.7 shows the throughput mean and throughput standard deviation for heuristic 1. This heuristic has better mean throughput than any of the non-skipping assignments shown in Fig. 6.2, which has the same queue threshold of 70 calls. With the processor execution time matrix chosen for these simulations, the heuristic reduces to a static assignment for most of the simulation runs and the non-skipping assignment default is seldom executed. This can be seen clearly by

examining the CPU utilizations for each of the cases, which are all similar and show similar bottlenecks in the earlier processors of the array as before.

Heuristic 1 always performs much better (average of 600 to 1200 calls per second higher) than the previous schemes because it does not attempt to complete processing in a single pass. Since user delays are required at the second and sixth modules, assignments which have this as a requirement are sub-optimal; the assignments generated by heuristic 1 do not have this restriction and are better as a result.

A second heuristic (heuristic 2) is obtained by modifying heuristic 1 as follows:

Heuristic 2

```
if (better than average at processing the given module) then process
else
    if (the worst processor for processing the given module) then
        pass through without processing
    else
        use one of the 126 assignments (where processors cannot be skipped)
        /* this is the default non-skipping static assignment */
```

Fig. 6.8 shows the throughput mean and throughput standard deviation for heuristic 2. The results for this heuristic are still better than the default non-skipping assignments shown in Fig. 6.3, which has the same queue threshold value of 30, but the scheme has poorer throughput performance than heuristic 1. The main reason for this is the over-utilization of processor 1 (and sometimes processor 2). Processor 3 is under-utilized as a result; this can be seen in the lower portion of Fig. 6.8.

If an assignment is generated which requires processing to be completed in a single pass, but which allows processors to be skipped, better system throughput is anticipated. This does not occur as can be seen by comparing Fig. 6.9 with Fig. 6.3. Mean throughput is only marginally inferior for the worst cases and only marginally superior for the best cases. The standard deviations for throughput are also very similar. The results indicate that skipping of processors in

a static assignment, all else being equal, does not improve performance significantly if a restriction of single-pass processing is placed on the assignment.

A similar heuristic to heuristic 1 (heuristic 1A) has been simulated using all of the default assignments which allow processor skipping and is given by:

Heuristic 1A

```
if (better than average at processing the given module) then process
else
    if (the worst processor for processing the given module) then
        pass through without processing
    else
        use one of the 1001 assignments (where processors can be skipped)
        /* this is the default skipping static assignment */
```

The results (Fig. 6.10) are very similar to those already obtained with heuristic 1 (Fig. 6.7), with the maximum mean throughput decreasing marginally. As in Fig. 6.3, the smaller queueing threshold used in the simulation of Fig. 6.10 decreases the variance in the mean throughput.

In some of the previous experiments, queueing bottlenecks affect the system performance. If the size of the queue in each processor is used as a criteria of whether to process or not, performance can be improved further. Fig. 6.11 shows an increase in the throughput mean relative to the non-skipping default case (the horizontal axis here is now only a simulation index, not an assignment vector). Comparing with Fig. 6.2, which is the default non-skipping case with the same queueing threshold, the throughput variance is generally lower and the CPU utilizations are more evenly balanced.

If care is taken to select the best fixed assignment for the current job mix, as described in assignment scheme (7A), the performance improves. The set from which these optimal assignments are selected is restricted to include only the non-skipping cases derived in Section 5.4.1. The selection is made based on local information (the co-ordinate axis is only a simulation index). From a comparison of the throughput means of Fig. 6.3 with Fig. 6.12, it can be seen that optimal schedules are selected for most of the simulations of Fig. 6.12. Also, the variance

decreases by about two orders of magnitude and the CPU utilizations are more stable, indicating better load balancing.

If the selection is made based on job mixes which are determined by the headend processor and then relayed on to the other processors, as described in assignment scheme (7B), the performance improves further. The worst performance shown in Fig. 6.13 is better than that shown in either Fig. 6.3 or Fig. 6.11. The reason for this is that there are fewer cases where calls have modules which are passed through by all of the processors. This can happen, in the case of local decisions, when processors make independent decisions as to the 'best' assignment, which are inconsistent with each other because they see different job mixes. Note that the variance in throughput is lower and that CPU utilizations are higher, although load is still balanced well.

The overall results indicate that the best schedule varies with job mix and the loading of individual processors. It is also clear that sub-optimal schedules are generated if all of the modules are restricted to finish execution in a single pass through the array. Also, if this restriction is made, processor skipping does not increase performance either. The heuristics used in this chapter yielded more productive assignments, both because better use was made of the specialized processors and because this restriction was relaxed.

So far, delays in processing at the second and sixth modules has been assumed, causing calls to be returned back to the first processor of the array for processing after each delay period. The effect on performance with the addition or removal of these lengthy delays during processing is considered in the following section.

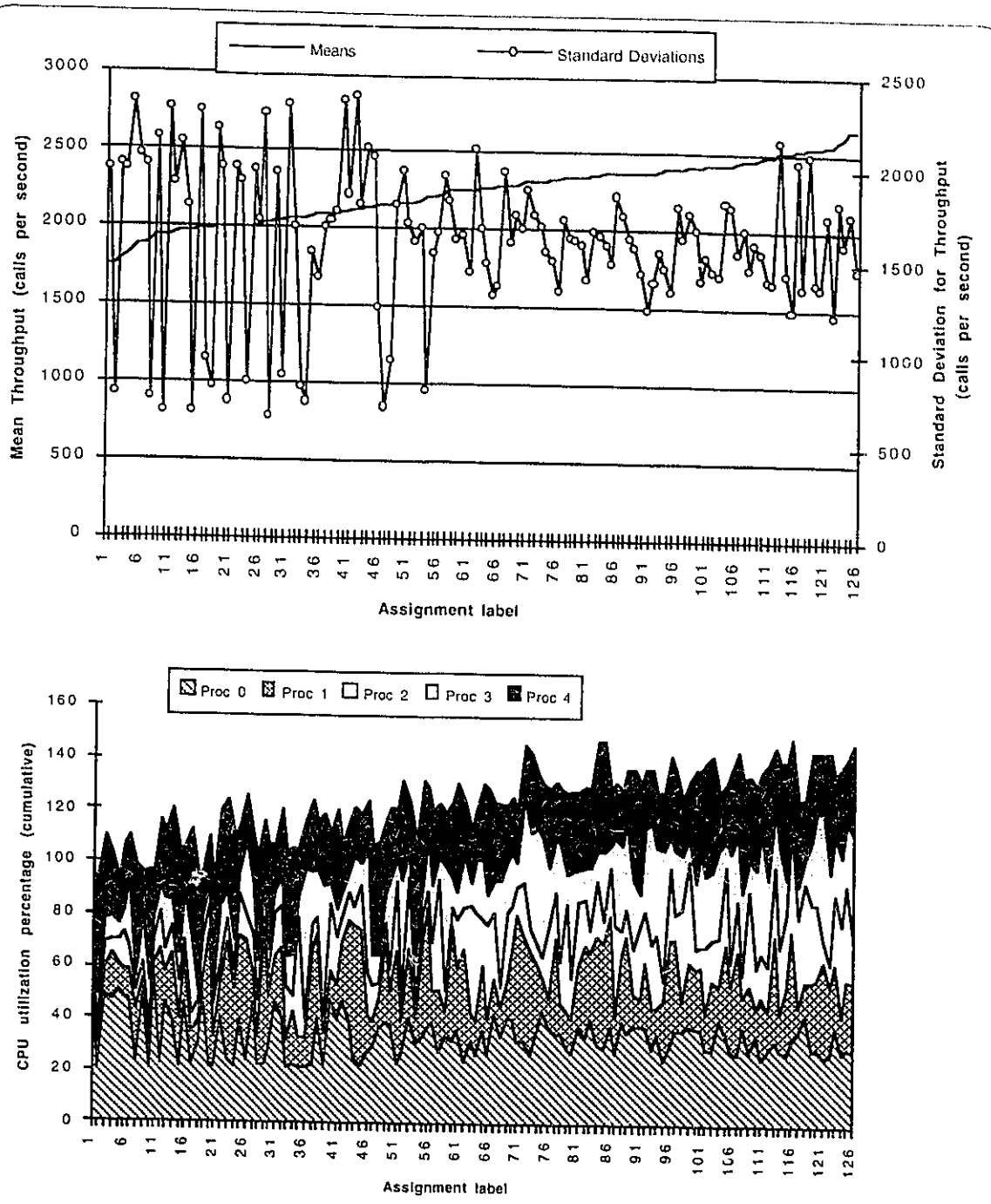


Figure 6.1 **Throughput & CPU Utilizations; no processor skipping (queue threshold of 150)**

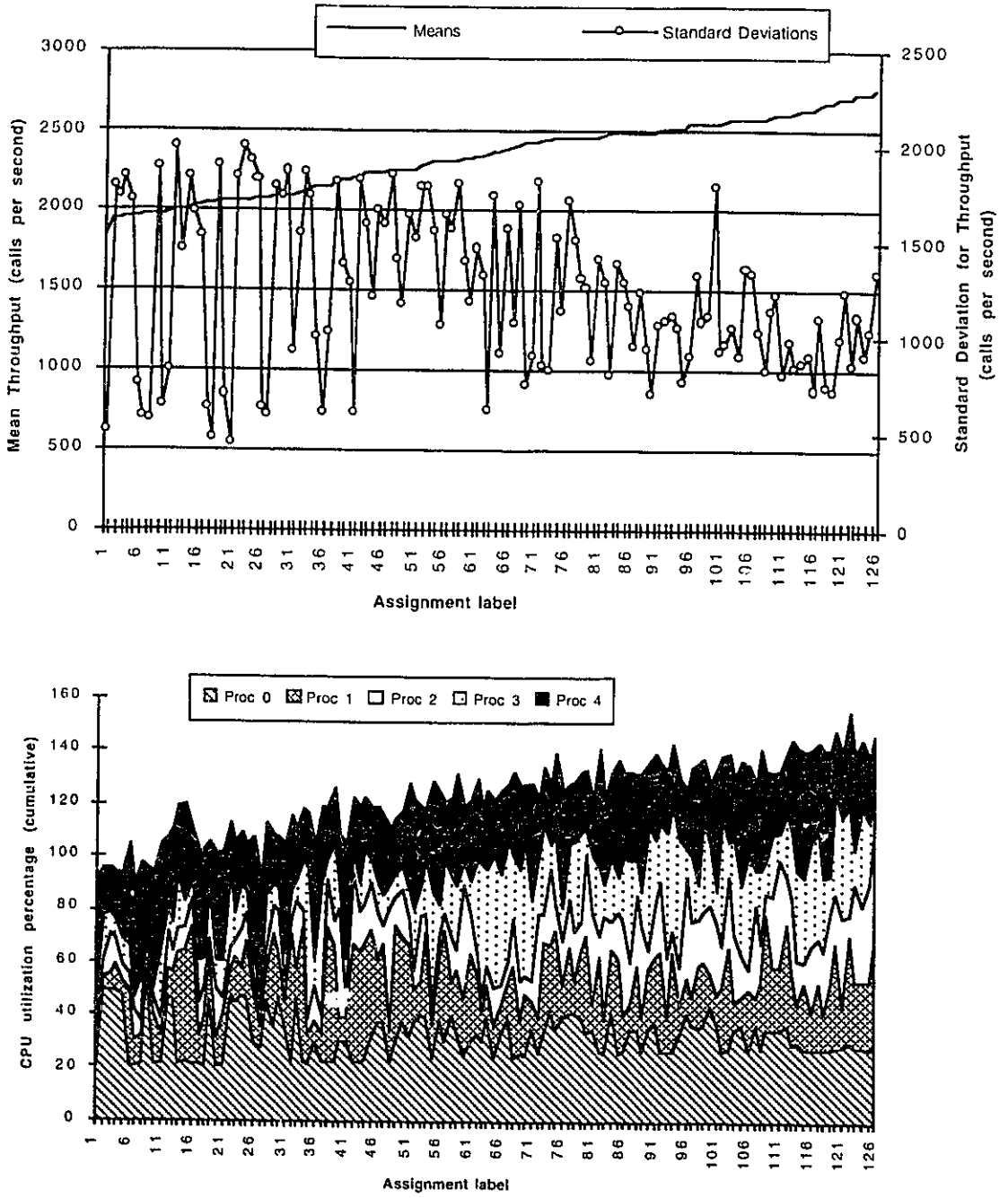


Figure 6.2 Throughput & CPU Utilizations; no processor skipping (queue threshold of 70)

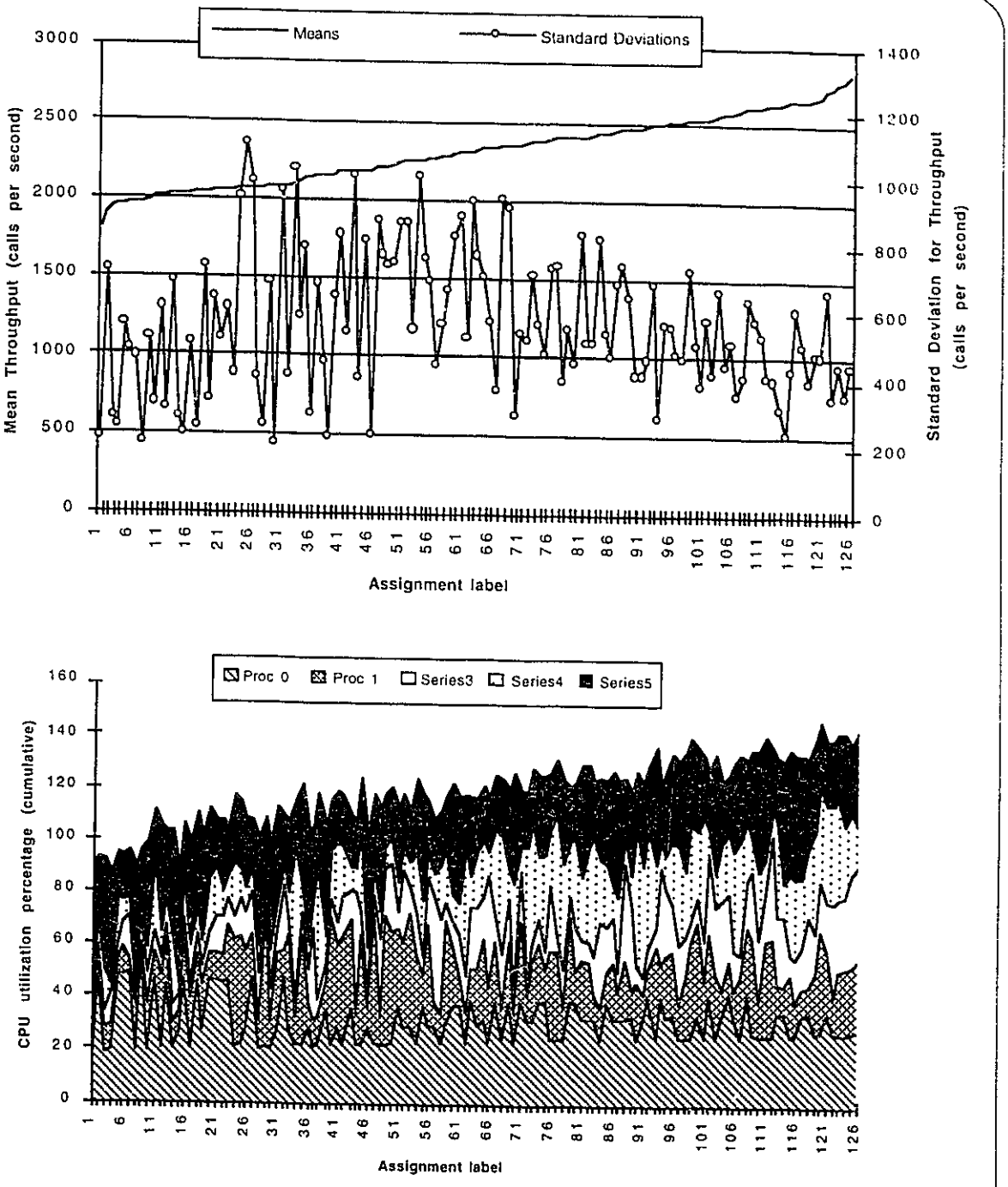
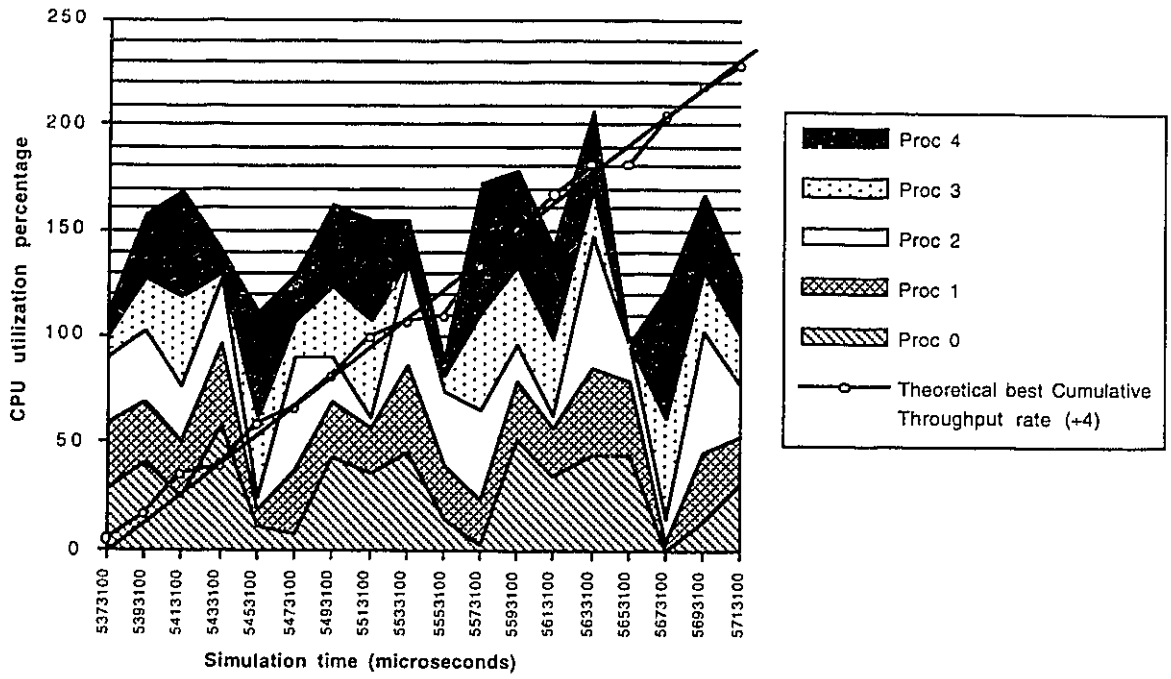
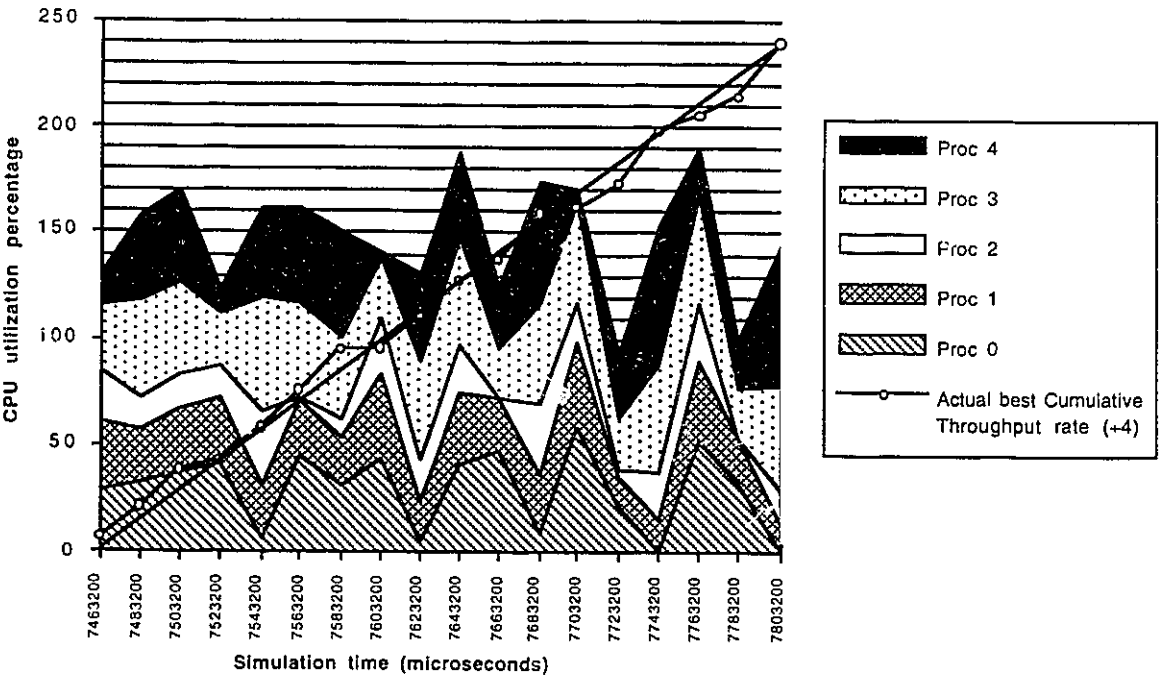


Figure 6.3 Throughput & CPU; no processor skipping (queue threshold of 30)

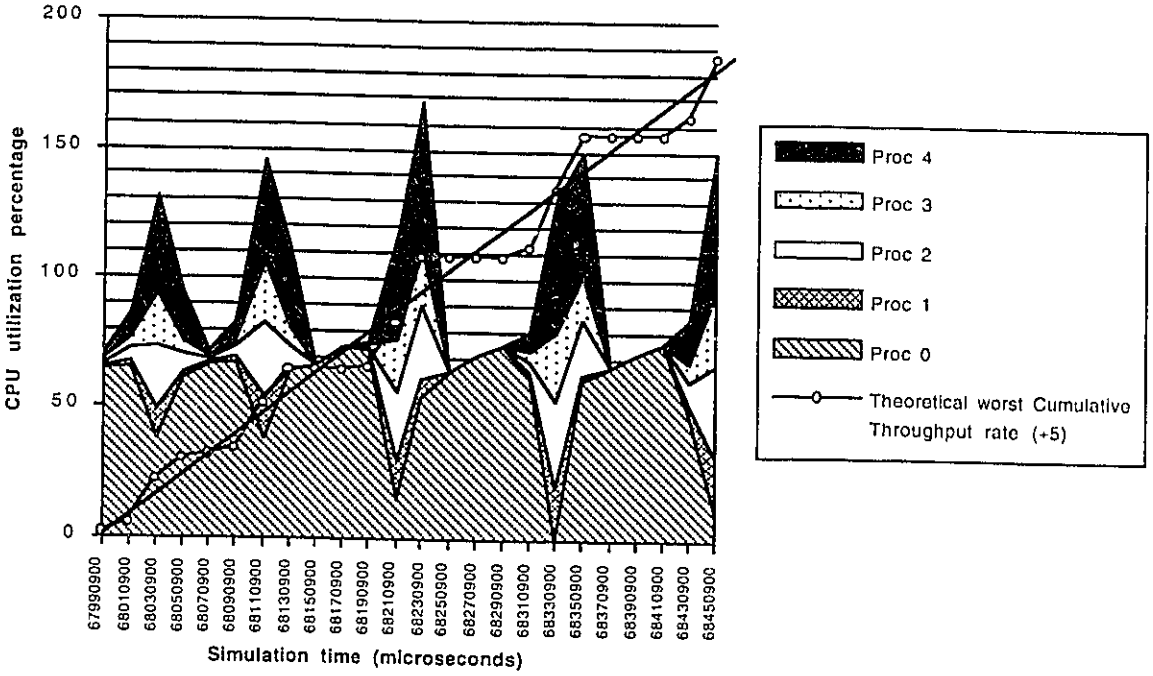


(B) Performance of the theoretically Best Assignment

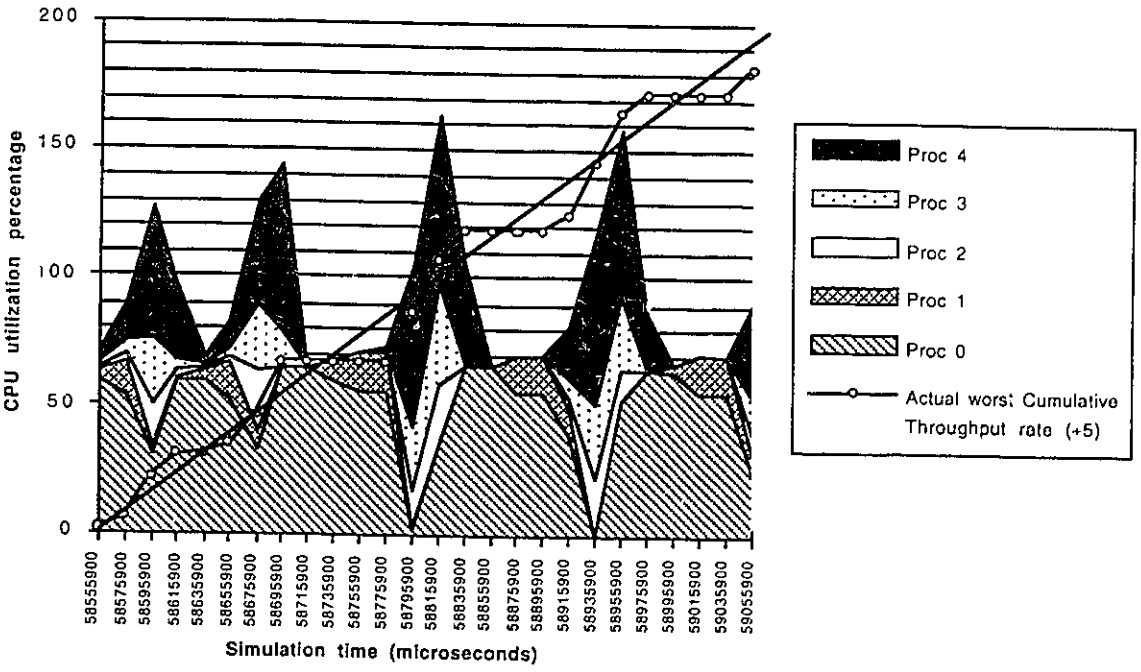


(A) Performance of the actual Best Assignment

Figure 6.4 CPU utilizations (actual and theoretical best cases; no processor skipping)



(B) Performance of the theoretically Worst Assignment



(A) Performance of the actual Worst Assignment

Figure 6.5 CPU utilizations (actual and theoretical worst cases; no processor skipping)

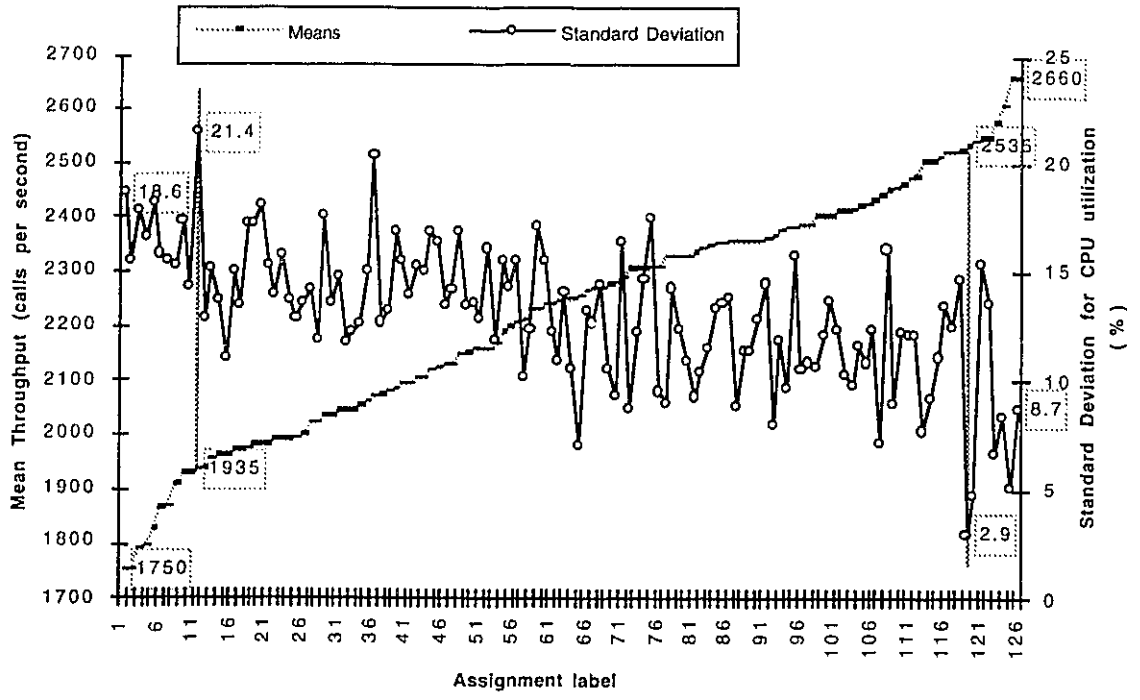


Figure 6.6 Processor CPU Utilization Variance and Mean Throughput; no processor skipping

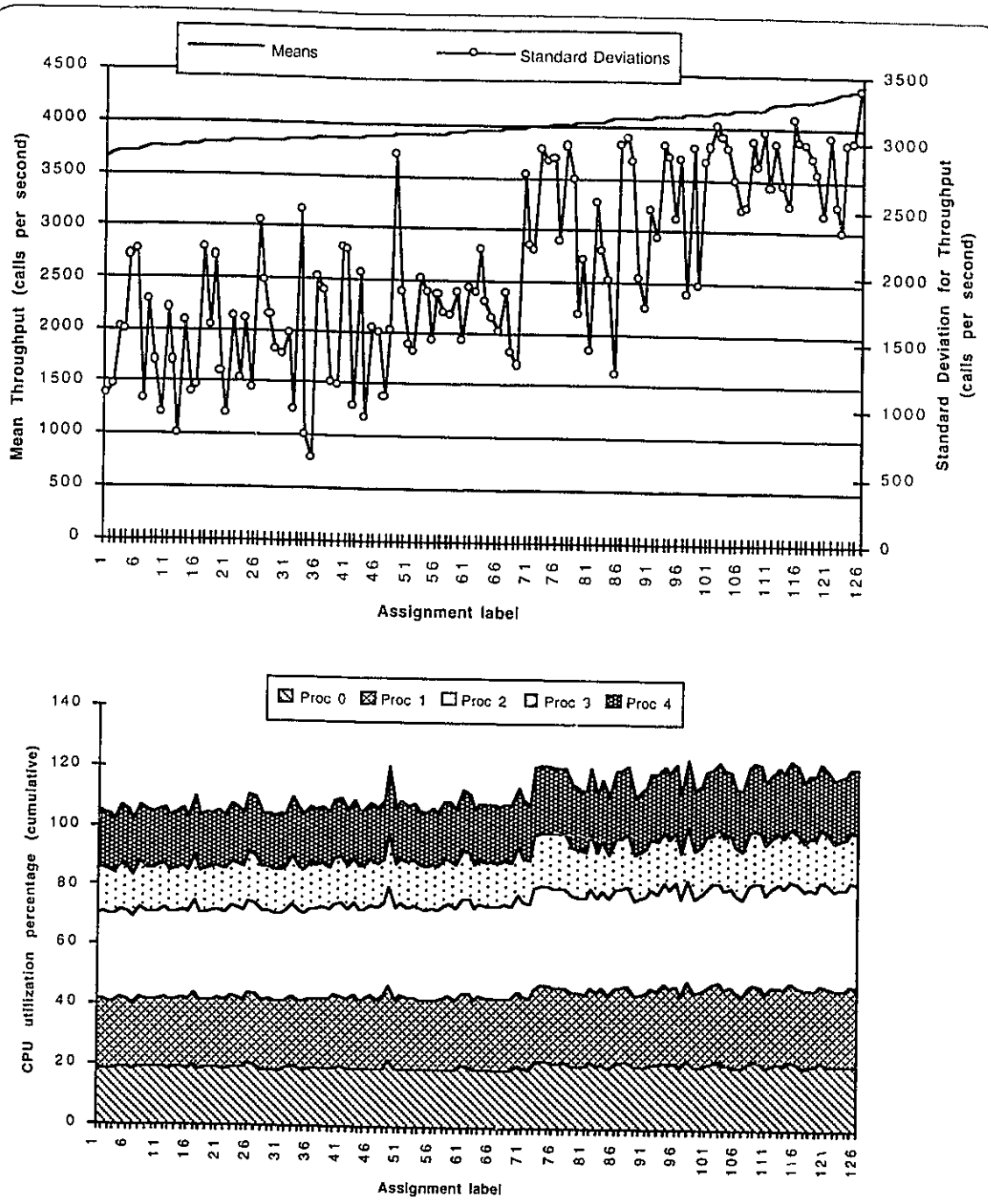


Figure 6.7 Heuristic 1: Throughput & CPU Utilizations (queue threshold of 70)

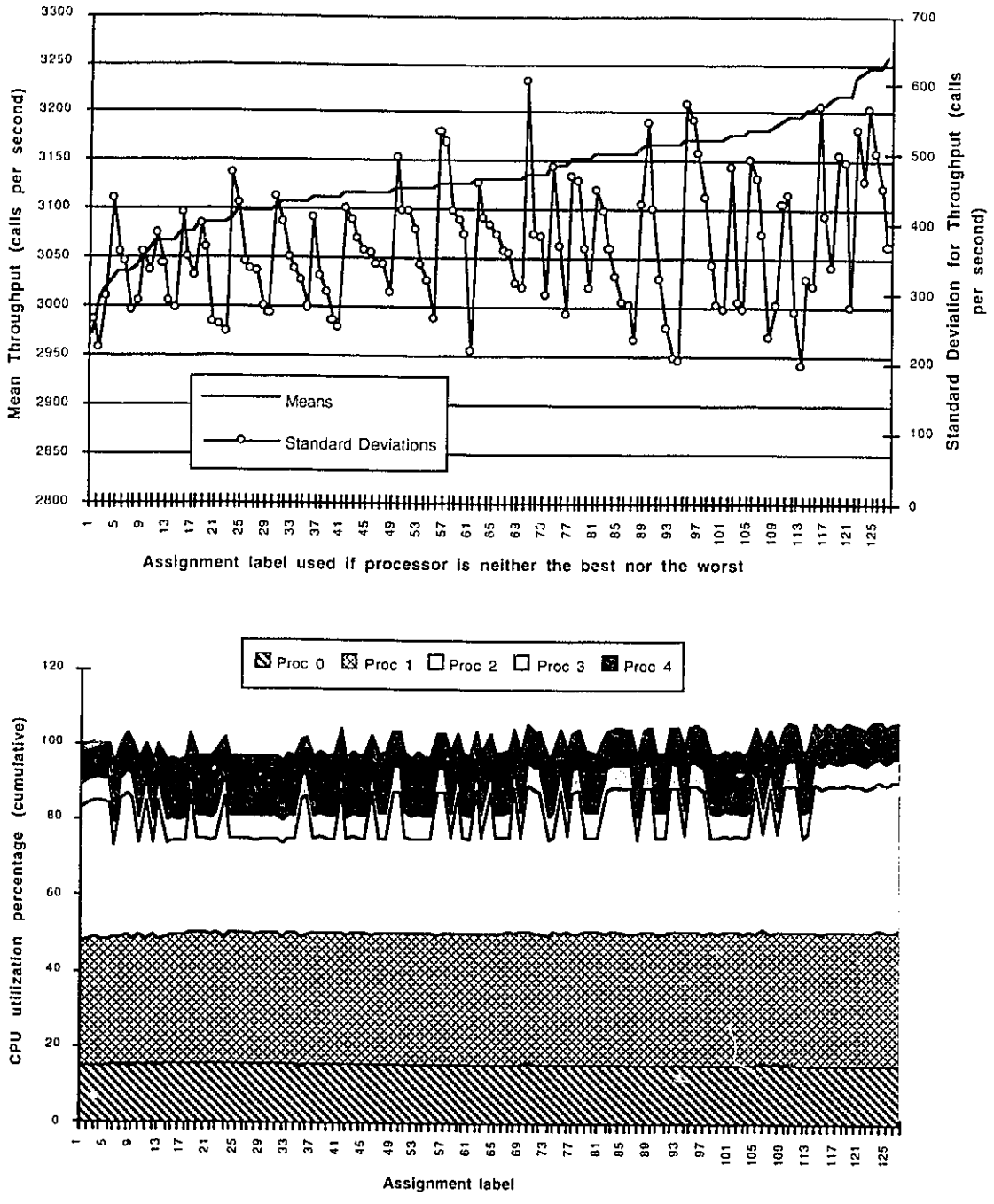


Figure 6.8 Heuristic 2: Throughput & CPU Utilizations (queue threshold of 30)

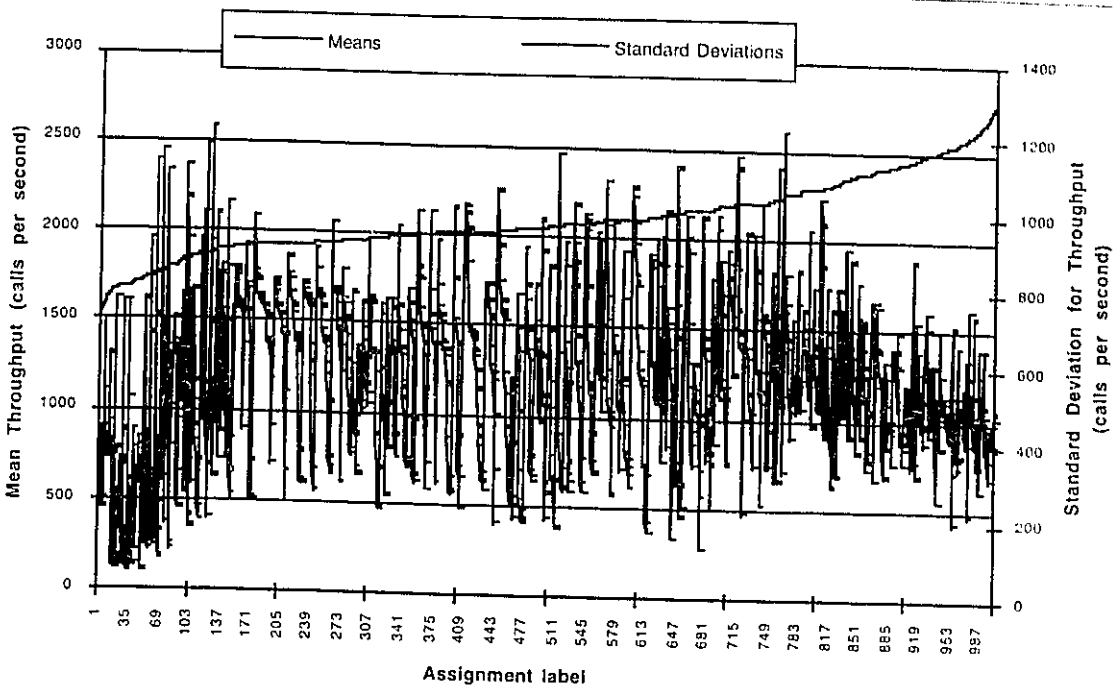
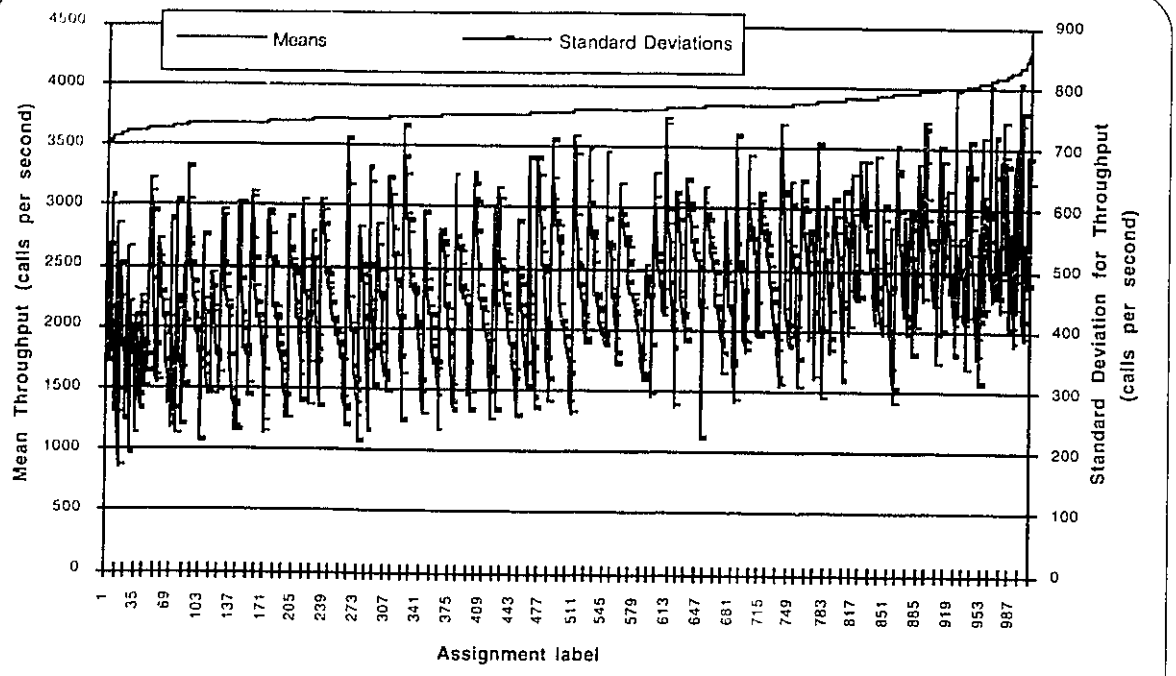


Figure 6.9 Throughput & CPU Utilizations; processor skipping (queue threshold of 30)



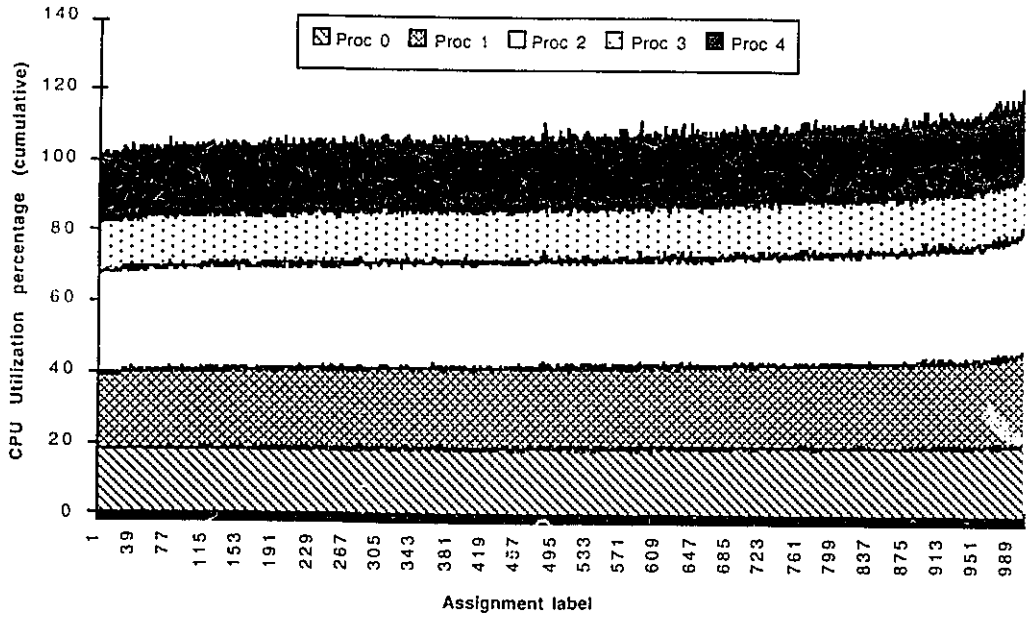


Figure 6.10 Heuristic 1A: Throughput & CPU Utilizations (queue threshold of 70)

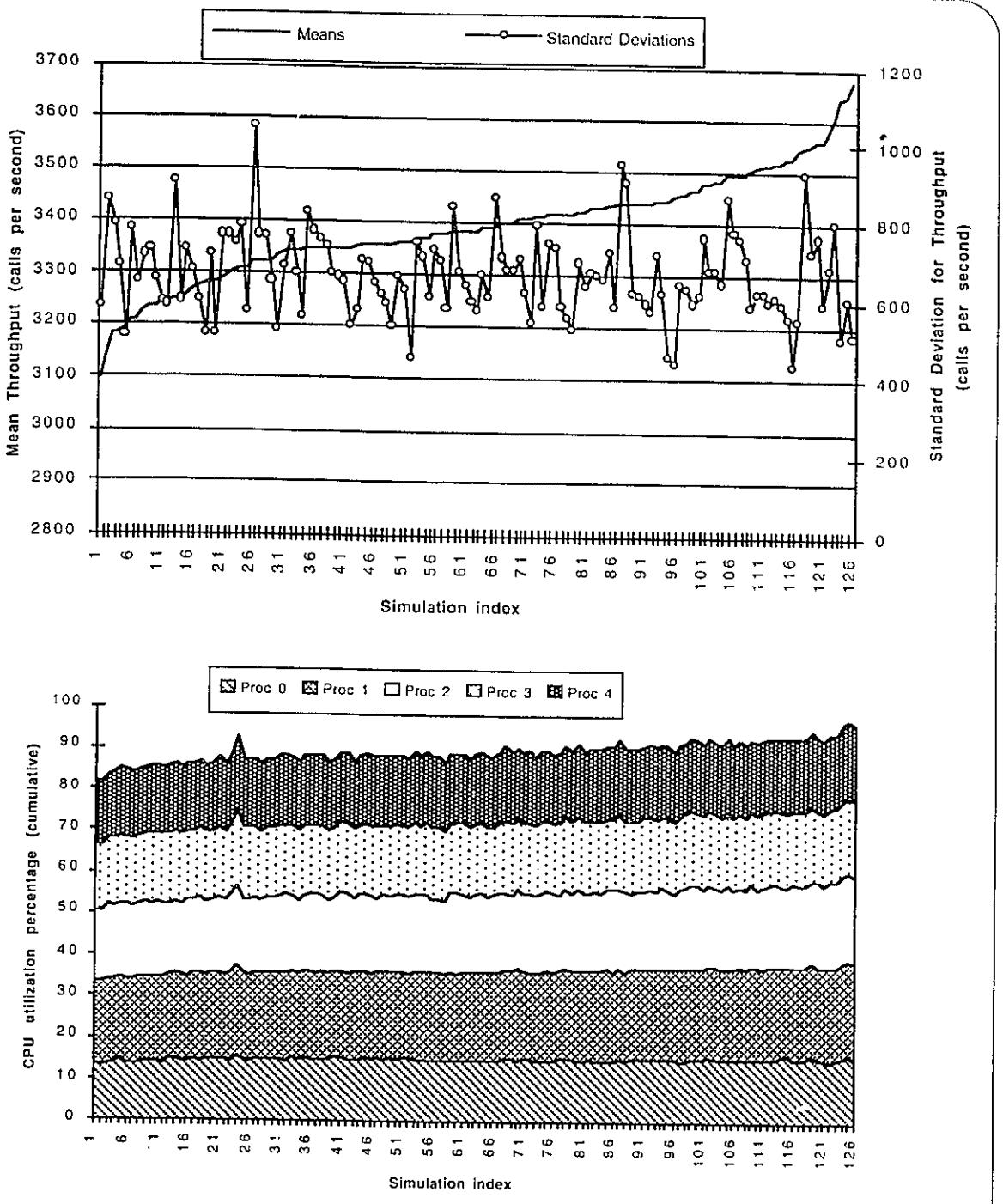


Figure 6.11 Throughput & CPU Utilizations; passthrough if busy (queue threshold of 70)

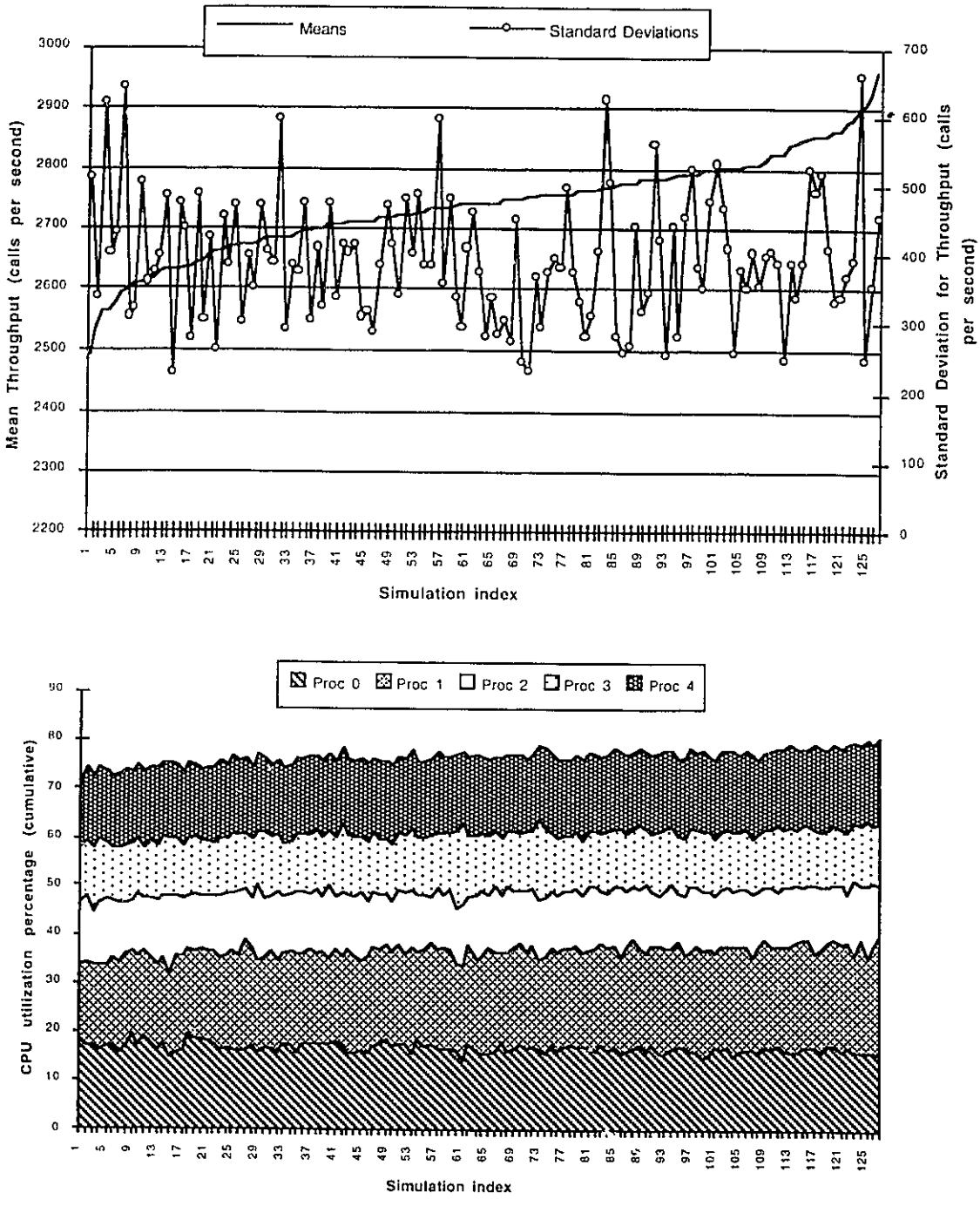


Figure 6.12 Throughput & CPU Utilizations; optimal local selection; no processor skipping

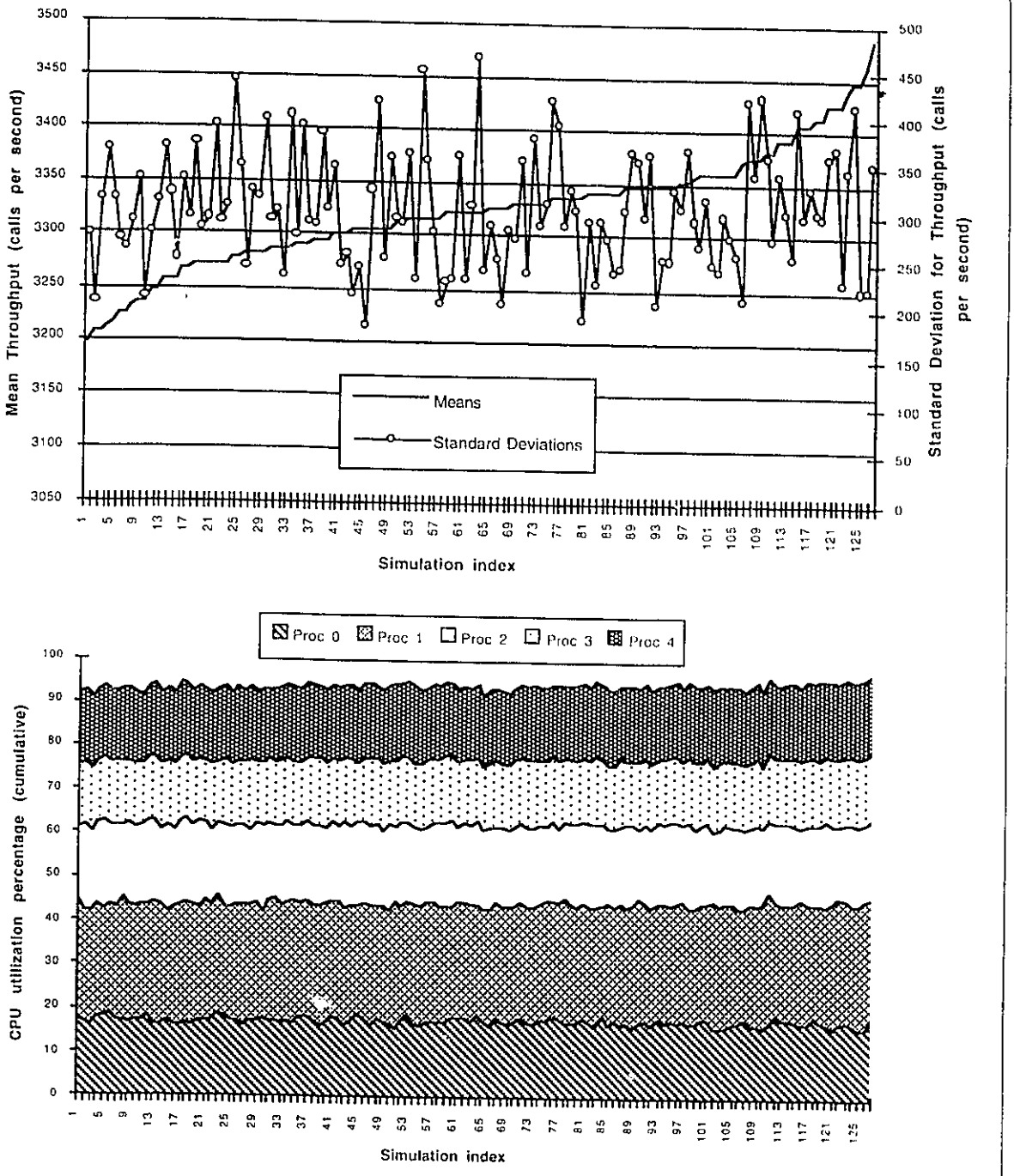


Figure 6.13 Throughput & CPU Utilizations; optimal global selection; no processor skipping

6.2 User Delays

All of the simulations performed in Section 6.1 assume a user delay with a mean of 50 μs which is exponentially distributed. If sub-calls are processed in a single pass through the array, without requiring any user interaction, the effect of user delay can be examined independently of the assignment scheme. All 126 possible assignments which do not permit processor skipping, but which can complete processing of all of the 10 sub-calls/modules in a single pass through the array, are simulated without user delays (Fig. 6.14).

(Fig. 6.14) can be compared directly with (Fig. 6.2). Both have the same input traffic rate of one new call being generated every 50 μs and the same threshold of 70 calls for each processor queue. The removal of the user delay has the immediate effect of reducing the variance of the throughput. This is because the queues in each processor are basically empty during the simulation. With user delays, calls which still require processing have to compete with newly-initiated calls as well as passing through the queues of different processors several times. The extra queueing delays increase the uncertainty associated with the processing of the sub-calls which are separated by delays to acquire user data.

The mean system throughput increases to the theoretical limit for the processor execution time array and the overheads assumed (see Section 5.3). The 'steps' in the graph of Fig. 6.14 are due to changes in the bottleneck point in the array. The big step increase at assignment 55 corresponds to the onset of end-of-line blocking in the array because of processors 3 and 4, which had previously been under-utilized.

In summary, user delays need to be modelled and have a large effect on call processing performance. Performance of the schedules which complete processing in a single pass through the array are dramatically improved if these user delays are removed.

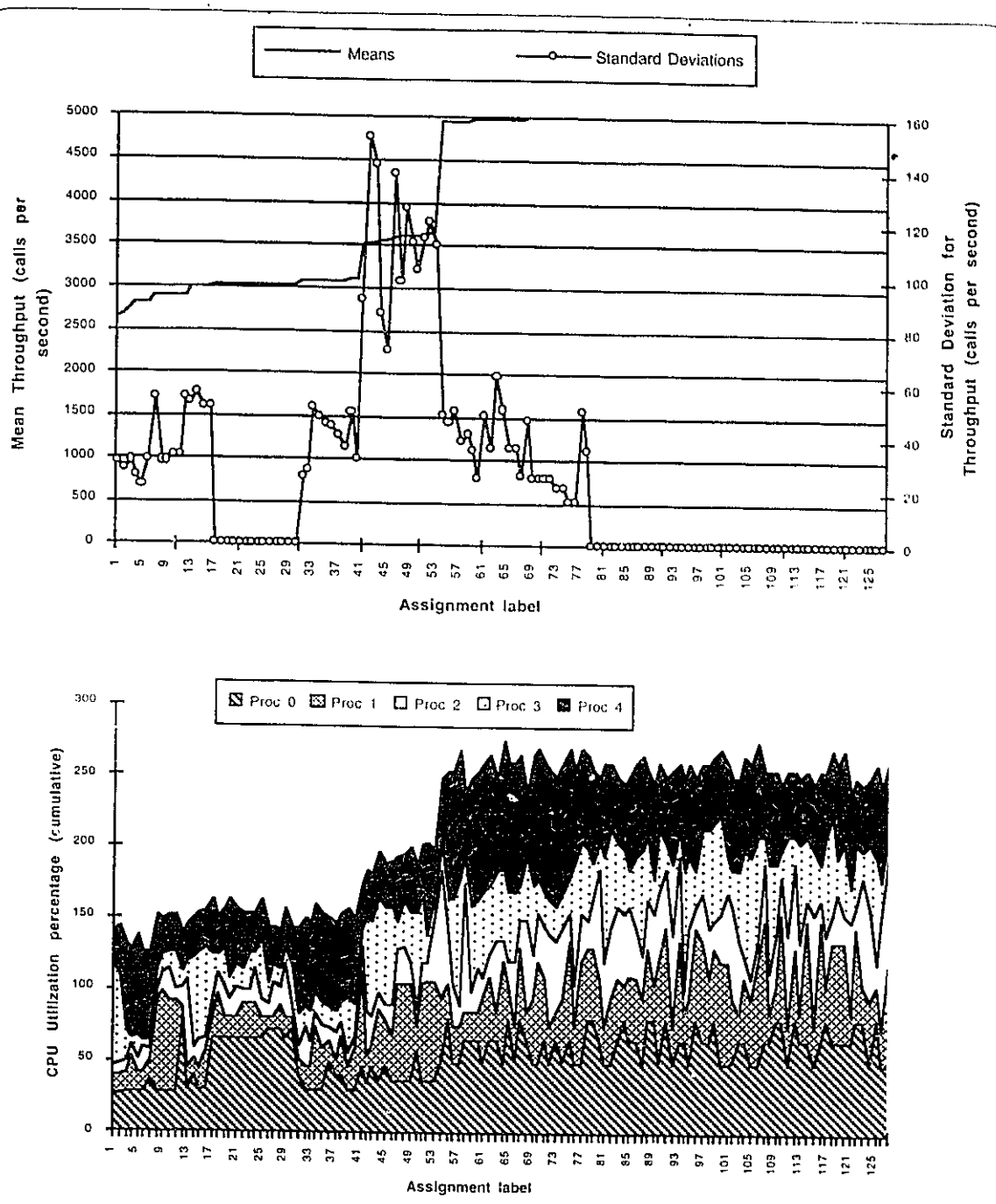


Figure 6.14 Throughput & CPU Utilizations; no processor skipping and no user breaks

6.3 Network Loading Conditions

An input traffic rate of one new call every 50 μs has been used for most of the previous simulations and it has been assumed that the architecture is the limiting factor. With traffic being generated at this rate, there are incoming calls which are being rejected (i.e. no dial tone given). To check the validity of this assumption, three simulations have been performed using non-skipping assignments and exponential delays of 25μs, 50μs and 100μs between generated calls.

The throughput mean and throughput variance for all of these are shown in overlaid form in Fig. 6.15 and the actual assignments for each case are given in Table 6.1. The results for both mean and variance are very close to each other, indicating that changing network load has minimal effect at these high overload levels (compare also with Fig. 6.3). The reason for this is that the same call rejection mechanism is used for all schemes, resulting in the same number of accepted calls; the number of rejected calls is, however, increased.

The call generation rate is decreased to 300μs between generated calls in Fig. 6.16. The mean throughput does not change, but the variance decreases to very low levels. Here, processing in the array is still the performance-limiting factor, but the latency associated with queueing of the calls has decreased substantially. If the call generation rate is decreased still further to 500μs (Fig. 6.17), the throughput variance decreases to nothing. However, mean throughput also decreases since performance is limited by the call arrival rate of 1/500 = 2000 calls per second.

In summary, varying the network load has no effect, provided the simple flow control mechanism is being used. The LCC assumption means that rejected calls disappear. For the LCH assumption, the network load can be expected to have more of an effect on the simulation results.

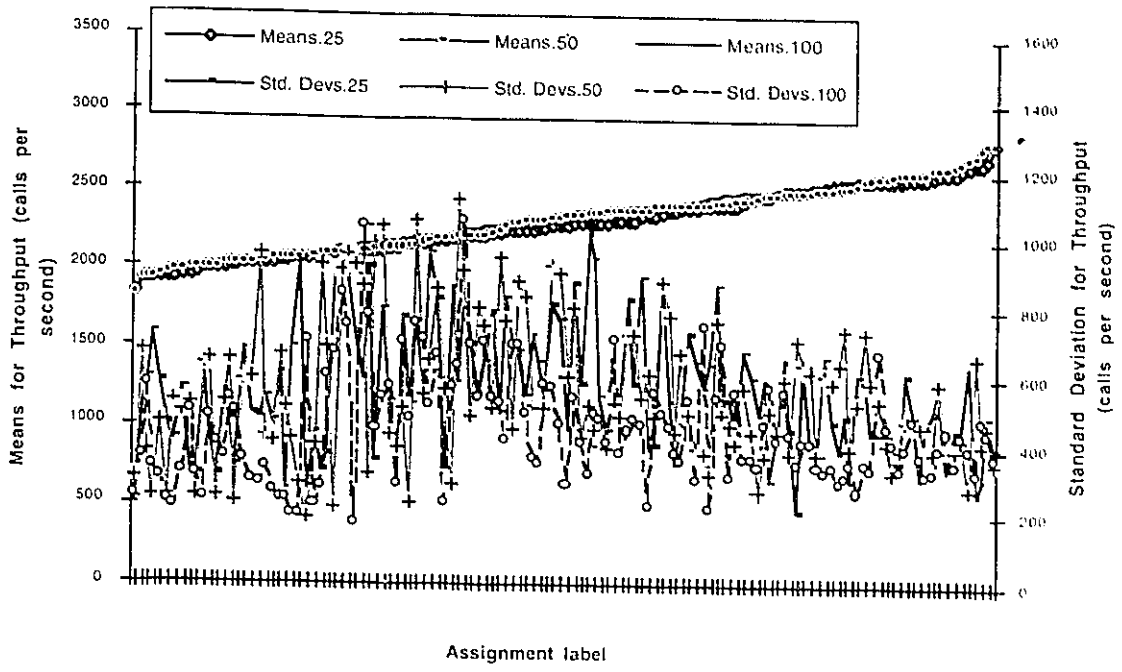


Figure 6.15 Throughput & CPU Utilizations: no processor skipping (different network loads)

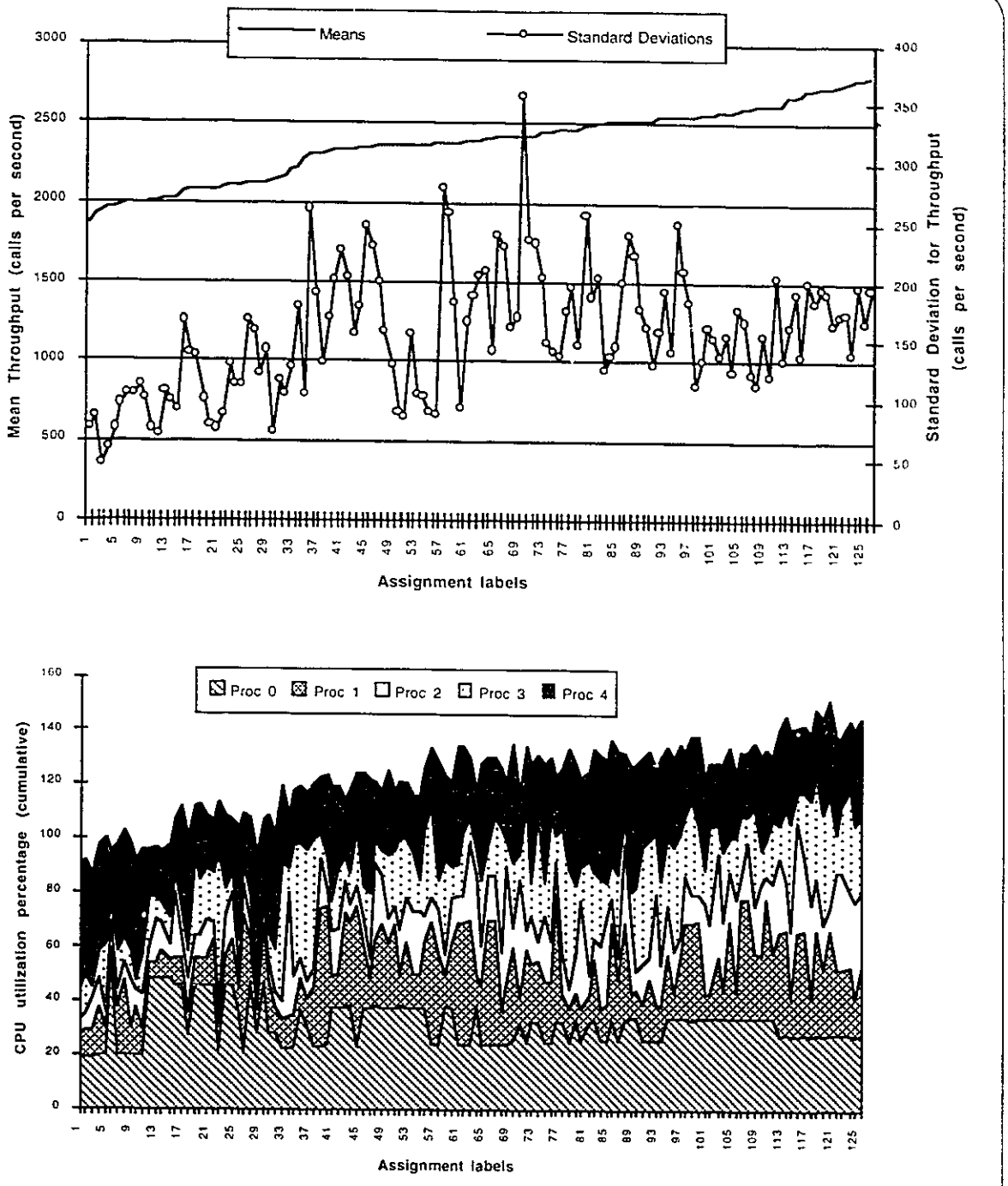


Figure 6.16 Throughput & CPU Utilizations; no processor skipping (300 μs between calls)

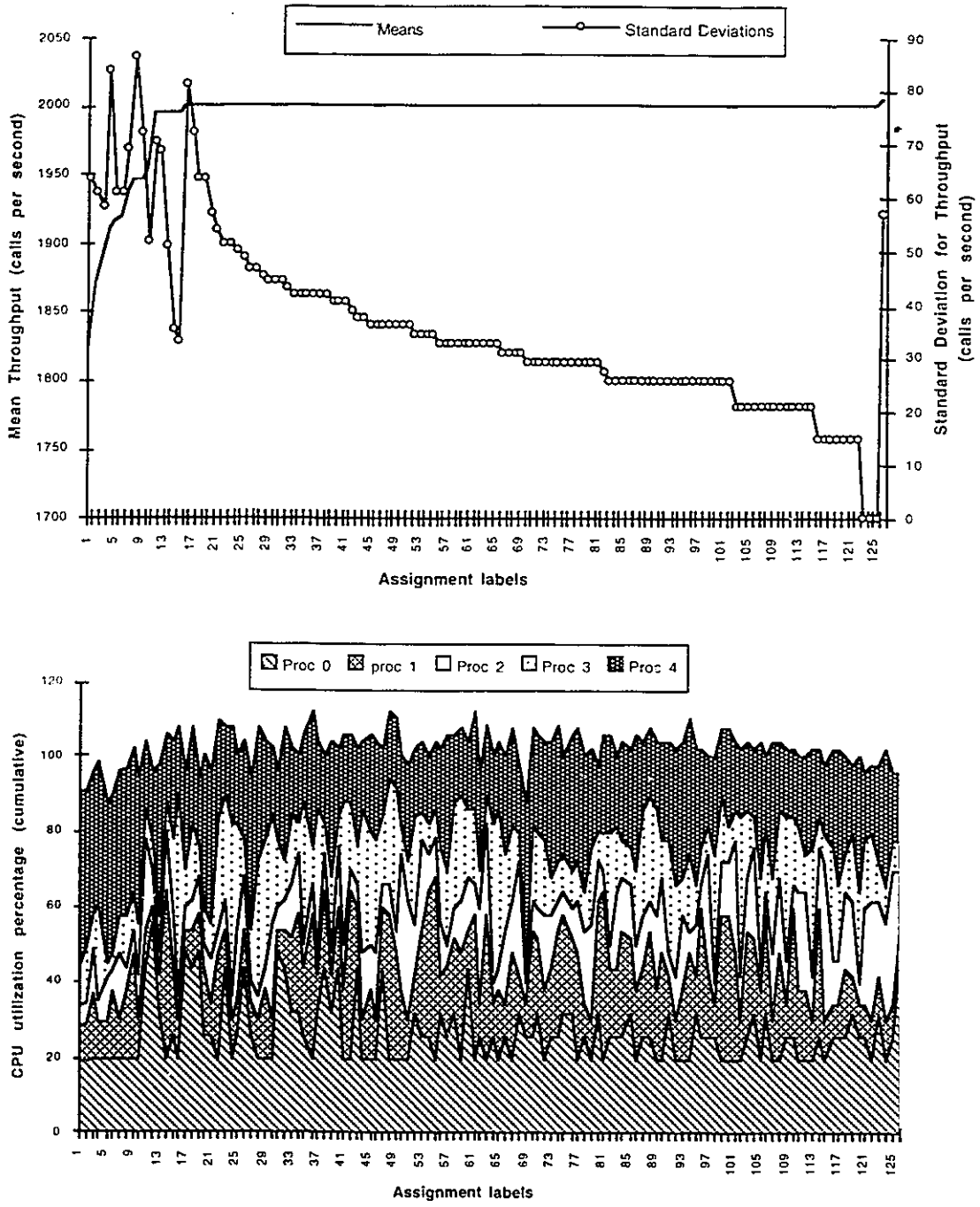


Figure 6.17 Throughput & CPU Utilizations; no processor skipping (500 μs between calls)

. Worst assignment labels			Mid-range assignment labels			Best assignment labels		
expo 25	*expo 50*	*expo 100*	*expo 25*	*expo 50*	*expo 100*	*expo 25*	*expo 50*	*expo 100*
			32122	33112	34111	13141	31411	31411
11116	11116	11116	15121	21511	14122	11422	21133 *	23221
52111	51112	51121	23212	15211	15121	31321	11341	11422
61111	13114	51112	33112	23122	23212	22132	11332	12241
51112	11125	13114	21214	15121	21124	12151	21142	21133
51211	61111	61111	15211	21124	11512	21223	23221	22312
13114	11215	11215	33211	32113	32113	13321	13141	31321
11125	51121	11125	24112	15112	32212	21142	31141	23131
51121	51211	51211	24211	11611	23311	22123	32311	21412
11215	52111	42112	11611	14221	31123	21241	12241	22213
12115	12214	52111	21511	14311	24211	11332	21232	21313
41122	42112	12115	15112	11521	14221	21322	13123	11431
42211	41131	41131	25111	22123	31132	13213	31321	13213
12214	12115	41122	23221	31123	21511	21232	13312	22123
42112	42121	11134	14221	32131	31312	13312	13132	13321
42121	41113	41113	31123	22312	23122	24121	13321	13123
21115	11314	41212	11521	31213	32131	22231	12142	21322
41113	41212	42121	14131	12511	11251	13222	21412	22132
43111	41122	41311	32131	32122	33211	12142	23131	12142
41311	23113	42211	11143	31231	11143	23131	21313	22222
41212	42211	41221	32212	32212	12511	22321	11323	22411
41221	11134	12214	23311	31312	32122	12313	21322	11332
41131	16111	43111	33121	11143	21151	13132	22321	12313
12124	41221	12124	31213	22213	33121	21421	13411	21232
16111	43111	11224	31231	11251	12151	22222	21322	21421
23113	21115	22114	31312	33211	31141	13123	13213	11323
11224	12124	16111	22213	33121	11521	21322	22231	13222
11134	41311	21115	12511	21151	31231	21313	21331	13231
11314	34111	11314	31222	32221	22141	21331	21421	21322
31114	11512	11161	31141	31222	31213	13411	21223	13132
23122	11224	15112	31411	31132	25111	12241	12223	13312
32113	23212	14113	22312	24121	31222	12223	22411	21331
34111	14131	14131	31132	23311	32221	12232	12133	21223
11512	14122	31114	11413	25111	24112	11323	11233	12223
11161	24112	33112	32311	24211	32311	11431	11431	22231
14122	31114	14212	21151	11242	24121	22411	12313	22321
22114	14113	11152	11341	13222	11341	13231	13231	12133
14113	14212	15211	11251	11422	13141	11233	22222	12322
11152	11161	14311	22141	21241	21142	12412	12232	12232
21124	22114	21214	32221	11413	11242	12331	12322	12331
14311	11152	23113	21412	22141	13411	12133	12331	11233
14212	21214	11611	21133	22132	21241	12322	12412	12412
32122	33112	34111	11242	12151	11413	12421	12421	12421

Table 6.1 Ordered Assignment Vectors for Different Call Generation Rates

6.4 Scaleability Issues

Most of the simulation work has been performed with five processors. However, as already explained, it is a requirement that the architecture can be scaled to use more processors for performance improvement or for the addition of new functions. Further simulations have been performed using eight processors using the assignments which can be completed in a single pass through the array with processor skipping (Fig. 6.18) and also without processor skipping (Fig. 6.21). The eight-processor architecture is made up of the same five processors in the same order with duplicates of the first three processors at the end; this has been done to allow comparisons to be made.

If processor skipping is allowed, the eight-processor architecture can perform better (Fig. 6.18) than the five-processor architecture (Fig. 6.9). The number of ways processing can be completed in a single pass increases from 1001 to 19,448. All of the assignments in the five-processor case are included in those derived for the eight-processor case; the mean throughput range for five processors is contained inside the result for eight processors, as expected. The eight-processor array mean throughput is about 20% higher than that of the five-processor architecture, which is poor/non-linear speedup.

The throughput variances of the two sizes of architecture are similar. Interestingly, the eight-processor array has a lower throughput variance for the worst and best assignment cases. There is better load balancing and more predictable delays because the queues are smaller in the eight-processor case. This can be seen by examining the CPU utilizations for the best and worst assignments (127 of each) in Fig. 6.20 and Fig. 6.19, respectively.

Section 4.1.4 noted that the throughput does not increase unless the bottleneck assignment changes. For the first 20 worst assignments, processor 4 is a bottleneck. Then, processor 3 becomes the gating stage. The big step increase in throughput variance (at assignment 69) is due to the beginning of program execution in processor 2 at the same time as processor 3, which remains the bottleneck.

For the better assignments, there is an increase in the uniformity and balancing of the loads. In general, heavy use is made of processors 1 and 2. Interestingly, processor 0 is always skipped; effectively, it is dedicated to accepting input from the network. However, the very best case uses only the last five processors, which corresponds to the assignment vector {0,0,0,1,2,2,3,2}. In fact, this is a different ordering of the basic set of five processors used for most of the other simulations. The bottleneck assignment depends on the particular processor/module execution time array as predicted in Section 4.1. Therefore, different processor orderings have not been explicitly investigated (see Section 5.5) to keep the results of different simulations as comparable as possible.

If processors cannot be skipped, there are only 36 ways of assigning 10 program modules to 8 processors. All of these cases have been simulated (Fig. 6.21) and the mean throughput improves, although the best case is only about 5% higher than that of the five-processor simulation results shown in Fig. 6.3. With these eight processors, the restriction of not skipping processors clearly improves load balancing over the CPUs, as can be seen from the CPU utilizations. Unlike the five-processor case, mean throughput can be 30% higher if processors are skipped (compare Fig. 6.21 with Fig. 6.18). The reason for this is that different processor orderings of a more efficient set of processors are possible, if skipping is allowed.

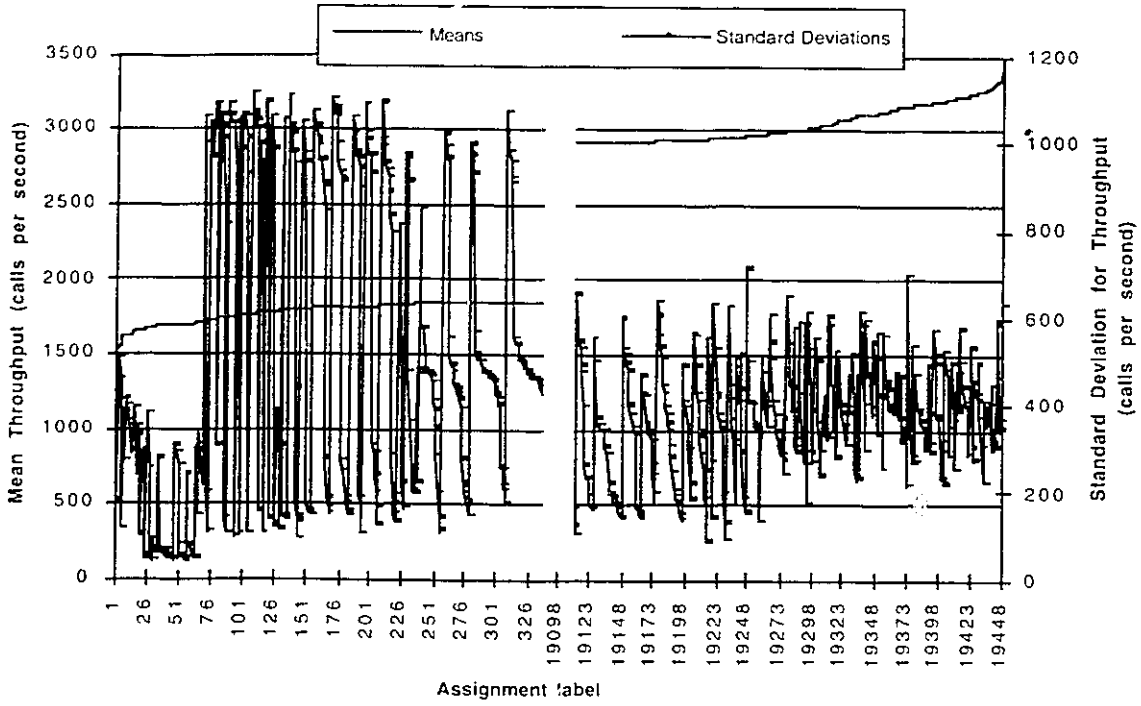


Figure 6.18 Throughput; best and worst cases; 8 processors; skipping allowed

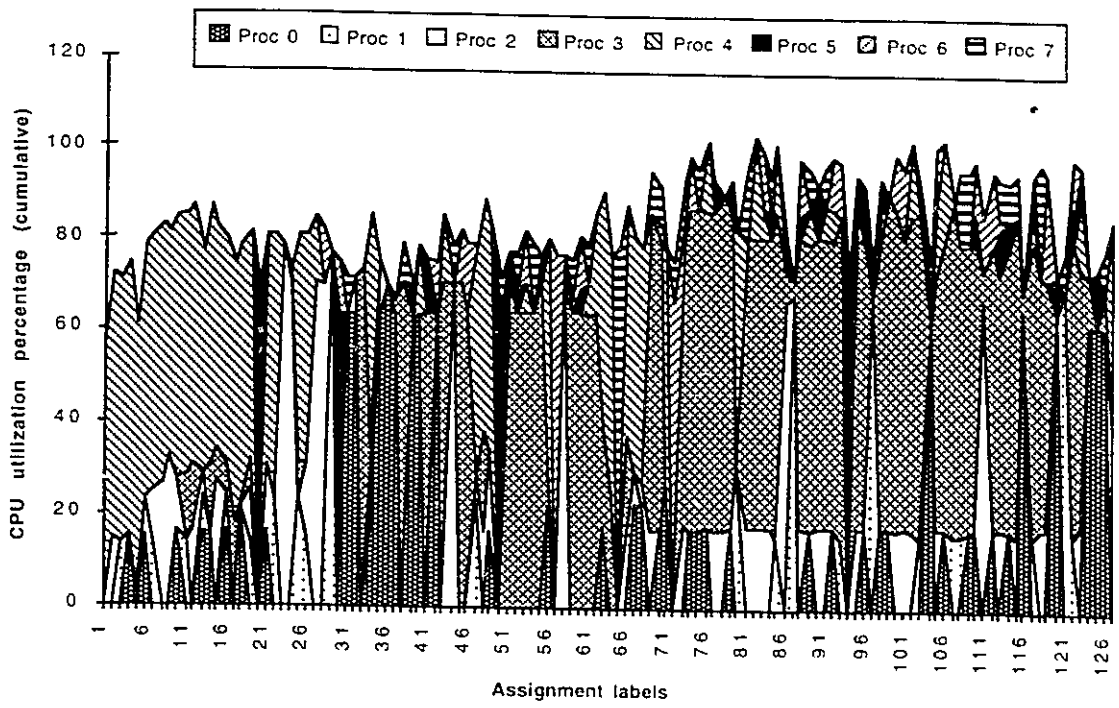


Figure 6.19 Worst 127 CPU utilizations; 8 processors and skipping

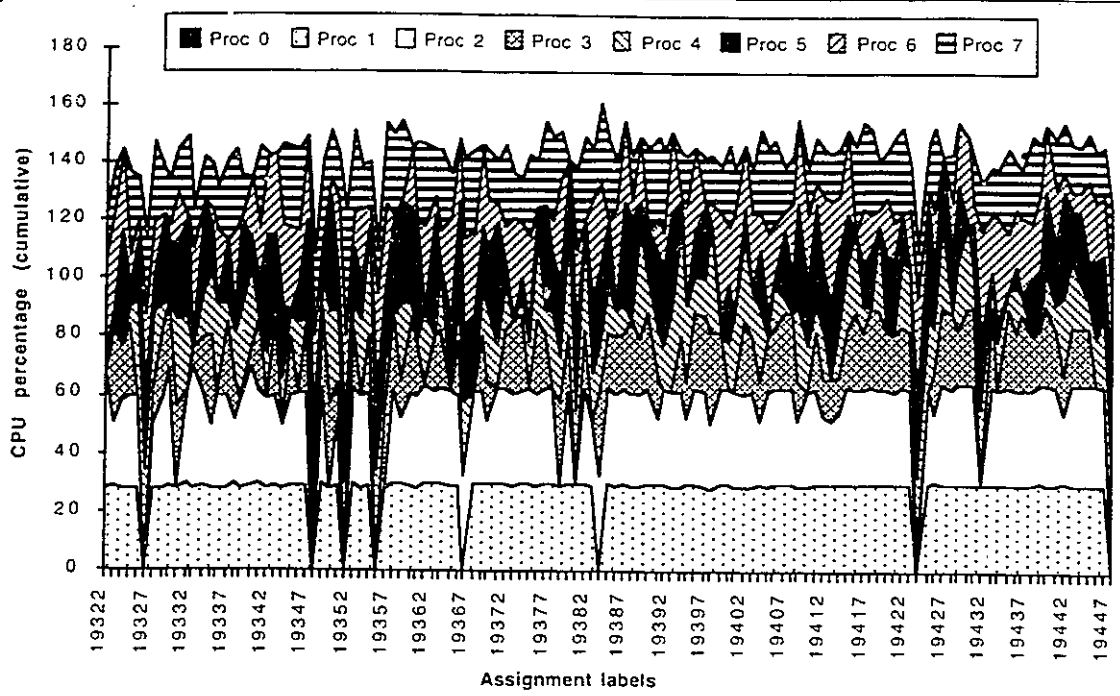


Figure 6.20 Best 127 CPU utilizations; 8 processors and skipping

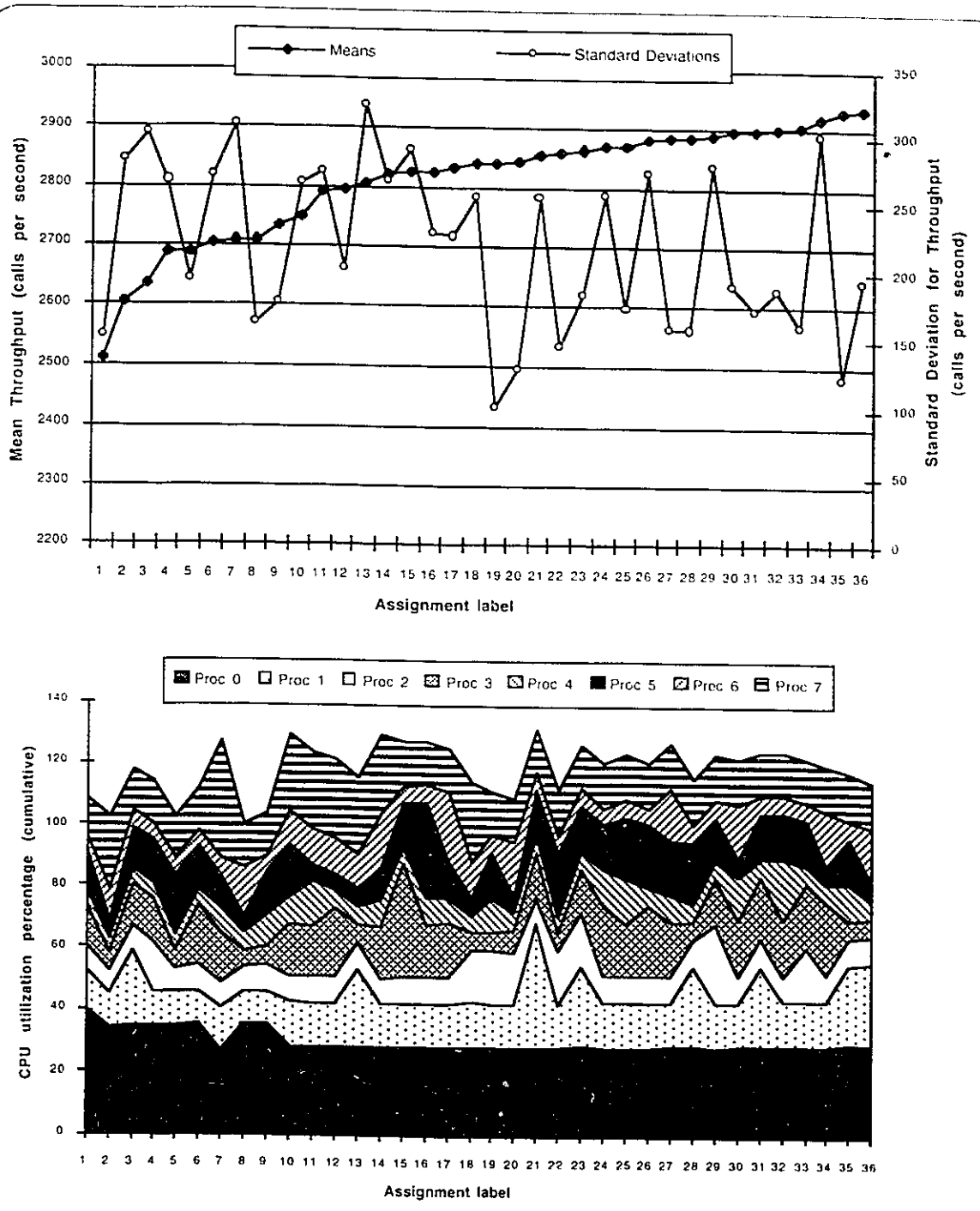


Figure 6.21 Throughput & CPU utilizations; 8 processors; no skipping (queue threshold of 30)

6.5 Summary

In this chapter, the simulation results for the Linear Pipelined Array Call Processing implementation have been described in detail. The assignment/scheduling constraint which restricts processing to be completed in a single pass through the array has been shown to limit system throughput. The presence of user delays also increased scheduling difficulty and decreased system throughput. The use of job or system state information improved throughput performance at the expense of increased scheduling complexity and increased communication requirements.

Variations in the network load had a limited effect on system throughput because of head-end queue saturation. Overload control and input traffic flow control at the head-end processor also limited system throughput, but the simple stop-and-wait algorithm used here works well. The Linear Array had poor scaleability and exhibited non-linear speedup as processors were added. Although performance improved with more processors, this was due mostly to improved load balancing across the array.

Chapter 7

Conclusions and Future Work

Call processing for a telecommunications switching system is a computationally demanding task which requires a multi-processor architecture to handle new telephone services and features. In this thesis, the design criteria for such a system have been summarized from the research and the most promising architectures evaluated against them. The complexity of distributed systems outweigh their performance advantages for the application.

A heterogeneous linear pipelined array has been proposed as a new type of call processing system and is a simple alternative to the more common load sharing architecture. A detailed comparison of the load sharing system and the linear array has been performed. The linear array has poorer scaleability, but can make better use of processor specializations, especially if access to busy centralized customer databases are required.

The linear array has been simulated extensively using the Verilog Hardware Description Language. The mean and standard deviation of the throughput are an excellent indication of the overall array performance. The individual CPU utilizations show the degree of load balancing in the array during a simulation and can be used to explain changes in the throughput statistics for different assignments of program modules to processors.

The effect of different assignments of program modules to processors has been presented. Poor performance is observed if the generated schedules are restricted to those which can complete processing in a single pass through the array. Better performance can be achieved using an appropriate heuristic rule which is not as restrictive.

If schedules are selected dynamically, based on the current ratio of calls in particular states, performance can be optimized further. This scheduling decision is best made on a global basis by the headend processor and is communicated down the array in the same direction as call processing takes place, which simplifies the co-ordination of schedule updates in each of the processors. Also, a disadvantage of generating schedules locally is that schedules may not be consistent with each other, causing processing of certain modules to be abandoned during these periods of inconsistency.

The method used to reject or accept calls into the system affects throughput. A simple algorithm which rejects input traffic when input queue sizes exceed a specific threshold has been used and works well. This technique allows comparison between different scheduling policies, since the architecture can adapt to process as many calls as possible, regardless of the scheduling policy being tested. Smaller queue thresholds result in less statistical variation in the throughput because it is less likely that processors are starved for input waiting behind congested upstream processors.

For optimum load balancing and performance, the best and worst assignments predicted by theory correlate very well with the measured CPU utilization results. Discrepancies with the best and worst cases for throughput are caused, however, by small imbalances in the delays through different processor stages. Re-ordering the processors in the array may give better load balancing and throughput, but is not practical in deployed systems.

If load imbalances between processors in the array can be reduced, while allowing each processor to perform its specialty function, throughput is maximized. The linear pipelined array scales well, up to the point where the bottleneck assignment delay cannot be reduced further. Adding more processors increases latency in the array without increasing throughput.

Delays for user input, which are required for newer telephone services, degrade performance substantially and complicate the task of deriving the optimal schedule. Switches are designed to

be able to operate during the busiest period of operation. The linear array architecture is robust to changes in offered traffic load and is very stable under overload conditions.

A detailed simulation is an effective method to analyze the underlying causes for events. Further, the models can be refined and adjusted on an ongoing basis to investigate specific areas of interest. These areas of interest can change based on unexpected simulation results and are not necessarily known at the simulation design and coding stages. Similarly, the high level of detail allows individual calls to be tracked, which simplifies upgrading of the simulation code to measure the effect of new call types and/or call mixes.

7.1 Future Work

The linear pipelined array has been examined carefully and a similar detailed treatment of the load sharing architecture would be informative. Different call types and call mixes of call types can be simulated for the linear array. Although performance under overload conditions is the most significant, performance for lighter loads could also be studied further. In particular, different and more sophisticated input throttling mechanisms should be investigated.

The Lost Calls Cleared (LCC) assumption made in the simulations could be replaced with the Lost Calls Held (LCH) assumption, which is used in the analyses of many North American telephone switching systems. Given the importance of load balancing for the linear array and the need to improve scaling performance, more work is required to verify the claims made in [88, 89] about the addition of extra processing buffer stages. Alternatively, the use of multiple pipelines and the task of interleaving their results could be studied as an interesting alternative to a full load sharing architecture. Distributed systems and load sharing systems promise to have more scaleable performance than the linear array, but are mainly limited by the requirement to access centralized and busy customer databases. Therefore, high-performance distributed database theory is an important area of further study.

A lot of work has been done in this thesis to investigate different scheduling schemes for the array, but the problem is NP-hard. Parallel Dynamic Interaction (PDI) is an inherently parallel

solution technique to this class of problems proposed recently [114, 121] and uses dynamic interplay between simultaneously executing sub-problems. PDI represents an exciting deviation from the more classical approaches [70, 68] and, therefore, seems like a promising approach worthy of further research and more detailed simulation. The homogeneous linear pipelined array scheduling problem has been solved efficiently in the literature and the heterogeneous linear array is a productive area of continued research for a high-performance implementation of the call processing function.

References

[1] M. H. Sherif, "Multimedia Networks and the Public Switched Telephone Network", *IEEE Communications*, pp. 92-94, January 1996.

[2] S. Srinivasan, "Impact of Enhanced Feature Interactions" *IEEE Communications*, pp. 100-102, January 1995.

[3] S. D. Personick, "The Evolving Role of Telecommunications Switching", *IEEE Communications,* pp. 20-24, January 1993.

[4] K. Stordahl and E. Murphy, "Forecasting Long-Term Demand for Services in the Residential Market" *IEEE Communications*, pp. 44-49, February 1995.

[5] I. Ebert and P. Richards, "Technology's Role in Switching Systems Evolution", *IEEE Communications,* pp. 26-32, January 1993.

[6] DMS Capacity Engineering Manual, BCS35 and up, 297-1001-170 Standard 02.03, November 1993.

[7] S. D. Personick and W. O. Fleckenstein, "Communications Switching- from Operators to Photonics", *Proceedings of the IEEE* , Vol. 75, No. 10, pp. 1380-1403, October 1987.

[8] R. T. Boyd, "Multiple Microprocessor Control of Telecommunication Switching", *IEE Colloquium on 'Microprocessor Applications Requiring High Integrity and Fault Tolerance'*, pp. 1-3, London, England, October 11, 1982.

[9] G. W. Couturier and A. W. Thompson, "Distributed Processing Control Complexes for Telephone Switching Systems", *IEEE International Conference on Communications*, Vol. 2, pp. 71-75, Chicago, Illinois, 12-15 June 1977.

[10] R. G. Cooper, "The Distributed Pipeline" *IEEE Transactions on Computers*, Vol. C-26, No. 11, pp. 1123-1132, November 1977.

[11] C. J. Stehman, "Distributed Control of the 580 Digital Switching System", *IEEE EASCON '87*, pp.11-3A - 11-3H, Arlington, Virginia, 26-28 September 1977.

[12] T. F. La Porta, M. Veeraraghavan, P. A. Treventi and R. Ramjee, "Distributed Call Processing for Personal Communications Services" *IEEE Communications*, Vol 33, No. 6, pp. 66-75, June 1995.

[13] M. Veeraraghavan, T. F. La Porta and W. S. Lai, "An Alternative Approach to Call/Connection Control in Broadband Switching Systems" *IEEE Communications*, pp. 90-96, November 1995.

[14] A. E. Joel, "Public Digital Switching Systems", *IEEE Communications*, Vol. 21, No. 3, pp. 6-10, March 1983.

[15] C. J. Hughes, "Switching - State of the Art", *British Telecom Technology Journal*, Vol. 4, No. 1, pp. 5-19, January 1986.

[16] TR-TSY-000511, Issue 2, July 1987 (Section 11, LSSGR)

[17] "Computer Performance as a function of the speed, quantity and cost of the processors". Proceedings Supercomputing '89, pp 759-764.

[18] J. D. Northcutt and E. M. Kuerner, "System Support for Time-Critical Applications" *Computer Communications*, Vol. 16, No. 10, pp. 619-636, October 1993.

[19] A. L. Decegama, *Parallel Processing Architectures and VLSI Hardware*. New Jersey: Prentice-Hall, 1989.

[20] F. Gourdou, R. Larousse and M. Maisonneuve, "Organization of Call Processing in the E10 S System", *Proceedings of the Fifth International Conference on Software Engineering for Telecommunication Switching Systems,* pp.173-177, Lund, Sweden, July 1983.

[21] M. L. Le Gac, M. Ruvoen and Y. Samoel, "A New Step for Alcatel E10: OCB283 connection and control", *Commutation & Transmission*, Vol. 13, No. 2, pp.69-76, 1991.

[22] M. L. Le Gac, M. Ruvoen and Y. Samoel, "Alcatel 1000 E10: Local Area Networks and Distributed Architecture", *Commutation & Transmission*, Vol. 15, No. 1, pp.65-72, 1993.

[23] M. I. O. Ero, "Hardware Reliability and Performance of the 5ESS Electronic Switch", *IEEE MIDCON '87*, pp. 164-167, Chicago, Illinois, September 1987.

[24] R. C. Cheung and W. A. Montgomery, "Functional Multiprocessing in an Experimental Digital Switching Office", *Proceedings of the IEEE Third International Computer Software and Applications Conference,* pp.74-79, Chicago, Illinois, November 1979.

[25] H. Berndt, "A Multiprocessor Approach to Communication Switching System Control", *EUROCON '84*, pp. 51-53, Brighton, England, September 1984.

[26] H. P. Brant, "DMS SuperNode System Overview", *IEEE GLOBECOM '88*, Vol. 3, pp. 1193-1199, 1988.

[27] R. Wood, "DMS-100 Technology Evolution", *Telesis,* Vol. 10, No. 3, pp.6-9, 1983.

[28] R. Swan, "DMS-100 Family Evolution", *Telesis,* Vol. 10, No. 3, pp.2-5, 1983.

[29] E. Salomonson, "Experience from multiprocessing in AKE13", *Proceedings of the International Conference on Software Engineering for Telecommunication Switching Systems*, pp.92-95, Salzburg, Austria, February 1976.

[30] A. Ash et al., "AXE 10 Central Processors", *Ericsson Review*, Vol. 67, No. 1, pp. 2-11, 1990.

[31] J. Grisham and K. Taya, "NECs digital experience: Evolving the Integrated Digital Network", *Telephony,* pp 52-62, June 23, 1980.

[32] A. Kitamura et al., "NEAX61 Digital Switching System Toward Broadband ISDN", *IEEE GLOBECOM '86*, Vol. 1, pp. 339-343, Houston, Texas, December 1986.

[33] M. Yamashita and M. Akiyama, "A Data Flow Control Electronic Exchange DATAFLEX-1", *Electronics and Communications in Japan,* Part 1, Vol. 68, No. 5, pp. 654-661, 1985.

[34] B. K. Basak, "System Architecture of a FETEX-150 Switch", *Telecommunications,* pp. 7-11, April 1993.

[35] H. Takeichi, S. Saito and Y. Kamaigaki, "Enhancement of FETEX-150 Digital Switching System", *International Switching Symposium,* Vol. 2, pp. 34-38, Phoenix, Arizona, March 1987.

[36] R. Sugioka et al., "DTS-1 Digital Telephone System for a Toll Office", *FUJITSU Scientific and Technical Journal,* pp. 1-36, March 1981.

[37] A. Niwa et al., "Multiprocessor Control Hardware Design for DTS-11 Digital Toll Switching System", *Review of the Electrical Communications Laboratories,* Vol. 3, No. 5, pp. 792-801, September 1982.

[38] T. Yamada and S. Ogawa, "Fault Tolerant Multiprocessor for Digital Switching Systems", *Symposium on Fault-Tolerant Computing,* pp. 242-252, Chicago, Illinois, June 1989.

[39] H. Watanabe et al., "Operating System for VLSI Multiprocessor Systems", *Proceedings of the Fifth International Conference on Software Engineering for Telecommunication Switching Systems*, pp.167-172, Lund, Sweden, July 1983.

[40] B. B. De and H. B. Krakau, "Fault-Tolerance in a Multiprocessor, Digital Switching System", *IEEE Transactions on Reliability,* Vol. R-30, No. 3, pp.246-252, August 1981.

[41] D. A. Mnichowicz, "Application of Multi-Processors in a Stored Program Telephone Switching System", *IEEE International Conference on Communications*, pp. 74-76, 1980.

[42] D. Jackson and K. Patfield, "Impacts of Multiprocessing on GTD-5 EAX Call Processing and Operating System Software", *International Switching Symposium '81,* pp. 13A2/1-7, Verdun, Canada, September 1981.

[43] G. J. Owen, "Rollback- A Method of Process and System Recovery", *Conference on Software Engineering for Telecommunication Switching Systems*, pp. 118-124, Colchester, England, April 1973.

[44] Y. Tokita et al., "ESS Software Architecture for Multi-Processor System", *Proceedings of the Third International Conference on Software Engineering for Telecommunication Switching Systems*, pp. 132-136, Helsinki, Finland, June 1978.

[45] T. Aoyama et al., "Software and Data Structure for Multi-Processor Digital Switching System", *IEEE International Conference on Communications*, pp.313-316, Amsterdam, Netherlands, May 1984.

[46] J. A. Brakel and J. H. Hiemstra, "Structural Aspects of the TCP 36 multiprocessor control of telephony switching System", *International Switching Symposium '79,* part III, pp. 970-977, Paris, France, May 1979.

[47] GR-2859-CORE, Issue 1, "Switched-Based Feature Interaction Requirements", December 1995.

[48] L. Bermejo et al., "Service Characteristics and Traffic Models in a Broadband ISDN", *Electrical Communications,* Vol. 64, No. 2-3, pp. 132-138, 1990.

[49] GR-1298-CORE, Issue 1, November 1993

[50] TA-NWT-001123, Issue 1, May 1991.

[51] R. R. Mina, "The Theory and Reality of Teletraffic Engineering", *Telephony*, pp. 32-37, April 12, 1971.

[52] R. R. Mina, "A Review of Elementary Traffic Theory", *Telephony*, pp. 22-28, May 31, 1971.

[53] R. R. Mina, "Probability Theory as used in Telephone Traffic Engineering", *Telephony*, pp. 32-36, June 28, 1971.

[54] R. R. Mina, "The Mathematics of Telephone Traffic Distribution", *Telephony*, pp. 26-48, August 2, 1971.

[55] R. R. Mina, "Understanding the Erlang B Formula", *Telephony*, pp. 72-86, September 13, 1971.

[56] R. R. Mina, "Improved Loss Formulas for smooth and rough Traffic", *Telephony*, pp. 26-29, January 24, 1972.

[57] R. R. Mina, "How Valid is the Molina lost-calls-held trunking formula", *Telephony*, pp. 32-38, Mar 6, 1972.

[58] R. R. Mina, "Queueing theory and its use in Traffic Engineering", *Telephony*, pp. 27-32, April 10, 1972.

[59] R. R. Mina, "Traffic Engineering: graded multiples in theory and practice", *Telephony*, pp. 28-31, July 3, 1972.

[60] R. R. Mina, "More on the theory and practice of graded multiples", *Telephony*, pp. 29-35, July 10, 1972.

[61] R. R. Mina, "Basic Design of Multistage switching systems", *Telephony*, pp. 32-36, April 1973.

[62] R. R. Mina, "Calculating Congestion in link systems", *Telephony*, pp. 33-41, May 1973.

[63] R. R. Mina, "Calculating Switching Congestion", *Telephony*, pp. 29-35, June 1973.

[64] R. R. Mina, "Determining Trunk Quantities in Networks with alternate routing", *Telephony*, pp. 28-42, July 1973.

[65] R. R. Mina, "Metropolitan Alternate routing configurations", *Telephony*, pp. 40-49, August 1973.

[66] S. Tridandapani et al., "Low Overhead Multiprocessor Allocation Strategies Exploiting System Spare Capacity for Fault Location and Detection", *IEEE Transactions on Computers*, Vol. 44, No. 7, pp. 865-877, July 1995.

[67] Z. M. Wojcik and B. E. Wojcik, "Rough Grammar for Efficient and Fault-Tolerant Computing on a Distributed System", *IEEE Transactions on Software*, Vol. 17, No. 7, pp. 652-668, July 1991.

[68] H. El-Rewini et al., *Task Scheduling in Parallel and Distributed Systems*. Prentice-Hall, New Jersey, 1994.

[69] C. McCann et al., "A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors", *ACM Transactions on Computer Systems*, Vol. 11, No. 2, pp. 146-178, May 1993.

[70] J. A. Stankovic et al., "Implications of Classical Scheduling Results for Real-Time Systems", *IEEE Computer*, June 1995, pp 16-25.

[71] H. El-Rewini et al., "Task Scheduling in Multiprocessing Systems" *Computer*, Vol. 28, No. 12, pp. 27-37, December 1995.

[72] "Intelligent Networks: Dedicated to Services", *Electrical Communication*, Vol. 65, No. 1, pp.38-43, 1991.

[73] C. H. Lee et al., "Optimal Task Assignment in Linear Array Networks", *IEEE Transactions on Computers*, Vol. 41, No. 7, pp 877-880, July 1992.

[74] T.-H. Wu, "Cost-Effective Network Evolution", *IEEE Communications*, pp. 64-73, September 1993.

[75] D. Bertsekas and R. Gallager, *Data Networks* Prentice-Hall, New Jersey 1987.

[76] G. R. Ash, "Dynamic Network Evolution, with Examples from AT&T's Evolving Dynamic Network" *IEEE Communications*, Vol. 33, No. 7, pp. 26-39, July 1995.

[77] P. Cronin, "An Introduction to TSAPI and Network Telephony", *IEEE Communications*, pp. 48-54, April 1996.

[78] T. Nixon, "Design Considerations for Computer-Telephony Application Programming Interfaces and Related Components", *IEEE Communications*, pp. 43-47, April 1996.

[79] M. A. Iqbal and S. H. Bokhari, "Efficient Algorithms for a Class of Partitioning Problems" *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 2, pp. 170-175, February 1995.

[80] Shahid H. Bokhari, "Partitioning Problems in Parallel, Pipelined and Distributed Computing" *IEEE Transactions on Computers*, Vol. 37, No. 1, pp. 48-57, January 1988.

[81] M. A. Iqbal, "Approximate Algorithms for Partitioning Problems" *International Journal of Parallel Programming*, Vol. 20, No. 5, pp. 341-361, 1991.

[82] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*. New York: McGraw-Hill, 1984.

[83] D. Knox and S. Panchanathan, "Parallel Searching Techniques for Routing Table Lookup", Proceedings of IEEE INFOCOM '93, pp. 1400-1405, March, 1993, San Francisco, CA, U.S.A.

[84] R.W.S. Tredigo and A. C. Downton, "Scalable Parallelism for Embedded Vision Applications: the Generalized Tree Pipeline" in *Applications of Transputers 3, Proceedings of the 3rd International Conference on Applications of Transputers*, pp. 383-387, August 1991.

[85] C. R. Vick, *Dynamic Resource Allocation in Distributed Computing Systems*. Michigan: UMI Research Press, 1981.

[86] A. C Downton et al., "Top-Down Structured Parallelisation of Embedded Image Processing Applications", *IEE Proceedings of Vis. Image Signal Processing*, Vol. 141, No. 6, pp. 431-437, December 1994.

[87] C. T. Gray et al., "The Design of a High-Performance Scalable Architecture for Image Processing Applications", Proceedings of the International Conference on Application-Specific Array Processors, pp. 722-733, September 1990, Princeton, NJ, U.S.A.

[88] E. Smirni et al., "Performance Gains from leaving Processors Idle in Multiprocessor Systems", *International Conference on Parallel Processing*, Vol. 3, pp. 203-210, CRC Press, August 1995.

[89] D. C. McCrackin, "Eliminating Interlocks in Deeply Pipelined Processors by Delay Enforced Multistreaming" *IEEE Transactions on Computers*, Vol.40, No. 10, pp. 1125-1132, October 1991.

[90] K. G. Shin and Y-C. Chang, "A Reservation-Based Algorithm for Scheduling Both Periodic and Aperiodic Real-Time Tasks" *IEEE Transactions on Computers*, Vol. 44, No. 12, pp. 1405-1419, December 1995.

[91] A. El-Amawy and Y. C. Chang Tseng, "Maximum Performance Pipelines With Switchable Reservation Tables" *IEEE Transactions on Computers,* Vol. 44, No. 8, pp. 1066-1069, August 1996.

[92] F. Kasparec, "Towards a Special Multitasking Multiprocessor for Public-Network Telecom Applications", *1990 International Zurich Seminar on Digital Communications. Electronic Circuits and Systems for Communications,* pp. 243-255, 1990.

[93] H. G. Rotithor, "Taxonomy of Dynamic Task Scheduling Schemes in Distributed Computing Systems" *IEE Proceedings- Comput. Digit. Tech.* Vol. 141, No. 1, pp. 1-10, January 1994.

[94] G. G. Sigalov and A. G. Lopato, "Modeling and Optimization of the Parameters of a Synchronous Pipeline Processor" *Automatic Control and Computer Sciences,* Vol. 24, No. 2, pp. 42-47, 1990.

[95] D. M. Nicol and D. R. O'Hallaron, "Improved Algorithms for Mapping Pipelined and Parallel Computations"*IEEE Transactions on Computers,* Vol. 40, No. 3, pp. 295-306, March 1991.

[96] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms" *IEEE Transactions on Software Engineering,* Vol. SE-3, No. 1, pp. 85-93, January 1977.

[97] L. R. Ford and D. R. Fulkerson, *Flows in Networks,* Princeton University Press, New Jersey, 1962.

[98] S-J. Chang et al., "Evolving Switching Systems into the Information Age", *Proceedings of the 14th International Switching Symposium*, Yokohama, Japan, Vol. 1, pp. 263-267, October 1992.

[99] P. J. Walker, "Evolution of the International Switched Telephone Service", *Proceedings of the 1st National Conference on UK Telecommunications,* London, England, pp. 47-51, October 1992.

[100] E. H. Mednick, "Tutorial #2: Architectural Modelling in Verilog HDL", *2nd Annual International Verilog HDL Conference*, Santa Clara, California, U.S.A., March 1993.

[101] C. Harvey, "Performance Engineering as an Integral part of System Design", *British Telecom Technology Journal*, Vol. 4, No. 3, pp. 142-147, July 1986.

[102] R. H. Thompson, "Modelling a Multiprocessor designed for telecommunication systems control", *ICL Technical Journal*, pp. 119-130, November 1984.

[103] I. Onyuksel, "Throughput Analysis of Multiple-Bus Multiprocessor Systems with Simultaneous Possession of Common Resources", *Simulation*, Vol. 61, No. 3, pp. 176-184, September 1993.

[104] C. W. McCarron and C.-H. Tung, "Simulation Analysis of a Multiple Bus Shared Memory Multiprocessor", *Simulation*, Vol. 61, No. 3, pp. 169-177, September 1993.

[105] Cadence Design Systems, California, U.S.A. *Verilog-XL Reference Manual*, Version 1.6, March 1991.

[106] G. Gordon, *System Simulation* Prentice-Hall, New Jersey, 1978.

[107] M. Krym et al. "DMS-100 New Peripheral Overload Control Modeling", *IEEE International Conference on Communications*, Vol. 2, pp. 1111-1117, 1987.

[108] CM Call Timing and Capacity Modelling - BCS36, System Engineering Bulletin, PLN-1001-012, April 29, 1994.

[109] P. Agrawal and A. Ng, "Computing Network Flow on a Multiple Processor Pipeline", *IEEE Journal on Parallel and Distributed Systems*, Vol. 5, No. 6, June 1994, pp 653-658.

[110] B. Olstad and F. Manne, "Efficient Partitioning of Sequences" *IEEE Transactions on Computers*, Vol. 44, No. 11, pp. 1322-1326, November 1995.

[111] J. G. Kalbleisch, *Probability and Statistical Inference I*, Springer-Verlag, New York, 1979.

[112] R. K. Kincaid et al., "A Multistage Linear Array Assignment Problem" *Operations Research*, Vol. 38, No. 6, pp. 993-1005, November-December 1990.

[113] V. Aleksandrov and R. Andonov, "A Systolic Linear Array for the Knapsack Problem" *Parallel and Distributed Processing*, pp. 285-299, Vol. 17, 1991.

[114] I. Pramanick and J. G. Kuhl, "An Inherently Parallel Method for Heuristic Problem-Solving: Part I - General Framework" *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 10, pp. 1006-1015, October 1995.

[115] C. Harvey and P.H. Griffiths, "Controlling Overloads in Multiprocessor Systems (ESS)", *IEE Fourth International Conference on Software Engineering for Telecommunications Switching Systems*, pp. 167-171, July 1981.

[116] C. Nyberg et al. "Dynamical Effects in Control Systems (Telecommunication Traffic)", *IEEE GLOBECOM '92*, Vol. 2, pp. 778-781, 1992.

[117] F. Bonomi et al., "A Case Study of an Adaptive Load Balancing Algorithm", *Queuing Systems Theory and Applications*, Vol. 7, No. 1, pp. 23-49, October 1990.

[118] J. M. Kumar et al., "Load Balancing Algorithms for an Extended Hypercube" *IEE Proceedings- Comput. Digit. Tech.* Vol. 141, No. 5, pp. 298-306, September 1994.

[119] W. H. Beyer (Ed.), *CRC Standard Mathematical Tables*, Boca Raton: CRC Press, 1987.

[120] L. A. Crowl, "How to measure, Present and Compare Parallel Performance", *IEEE Parallel and Distributed Technology*, pp 9-25, 1994.

[121] I. Pramanick and J. G. Kuhl, "An Inherently Parallel Method for Heuristic Problem-Solving: Part II - Example Applications" *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 10, pp. 1016-1028, October 1995.

Appendix A

Basic Call Models

The following sections give a detailed description of the Originating and Terminating Basic Call models taken from [49, 50]. Detailed state diagrams are given for each model, where states represent essential operations required to establish, maintain and clear a channel. These states are referred to in [49] as Points-In-Call (PICs).

Legal transitions between PICs are given. Events causing transitions between PICs, detected by access processing (e.g. remote processors) or underlying Call Processing mechanisms in the core, are also presented. The Basic Call Models also include Trigger Check Points, which are key points where notification of events can be exported from Call Processing software. The Call Processing software can also import information from other processors, which can influence/direct subsequent processing after the Trigger Check Point.

The Basic Call Models cover the following types of access:

- non-ISDN user [49]: this could be voice-based POTS or 56kbps rate-adapted data
- ISDN user (Class I or II) [48]: ISDN users are identified by a directory number/ call type or by the actual physical interface
- Electronic Key Telephone Service (EKTS) [48]: group of ISDN Class I users (this group can also include a single non-ISDN user)
- conventional trunk: A trunk is a connection between switching systems (as opposed to a subscriber line). 'Conventional' describes public trunks which support traditional in-band signalling or exchange access in-band signalling. Multi-frequency, dial pulse or Dual-Tone Multi-frequency address signalling may also be supported.
- SS7 trunk: Signalling on these trunks uses the Common Channel Signalling scheme Signalling System number 7 protocol. This is a standardized protocol which allows switches (from different vendors) to establish connections for POTS or ISDN calls [5].

- private facility trunk: These are private dedicated trunks (e.g. tie trunk for dedicated tie line calls leased by an organization)

The previous list illustrates that the Basic Call Models are directly applicable to a broad range of applications. However, for the purposes of this document, the Basic Call Models described here provide information pertaining only to non-ISDN users. Even so, reasonable comparison and evaluation of different multi-processor implementations for Call Processing is still possible.

A.1 Originating Basic Call Model

The Originating Basic Call Model (Fig. A.1) covers the three basic stages of Call Processing [49] with corresponding PICs given for each:

- Call setup: NULL, AUTHORIZING ORIGINATION ATTEMPT, COLLECTING INFORMATION, ANALYZING INFORMATION, SELECTING ROUTE, AUTHORIZING CALL SETUP
- Stable Call: CALL PROCEEDING, WAITING FOR AN ANSWER, ACTIVE
- Call Clearing: RELEASE PENDING

The following tables give summarized descriptions of each of the PICs, using the following table headings:

- Action - A description of the Call Processing operation associated with the PIC
- Entry Event(s) - Event(s) causing the action to be initiated
- Exit Event(s) - Event(s) signifying completion of the action
- Exit Event(s) related to incomplete calls - Event(s) signifying the Call Processing associated with incomplete calls*
- Available Information - This is a list of available model information which is relevant to Call Processing applications.

*For example, if disconnected or cleared events are received before a call is answered, they would be classified as exit events related to incomplete calls. Otherwise, if they are received afterwards, they are treated as Exit Events

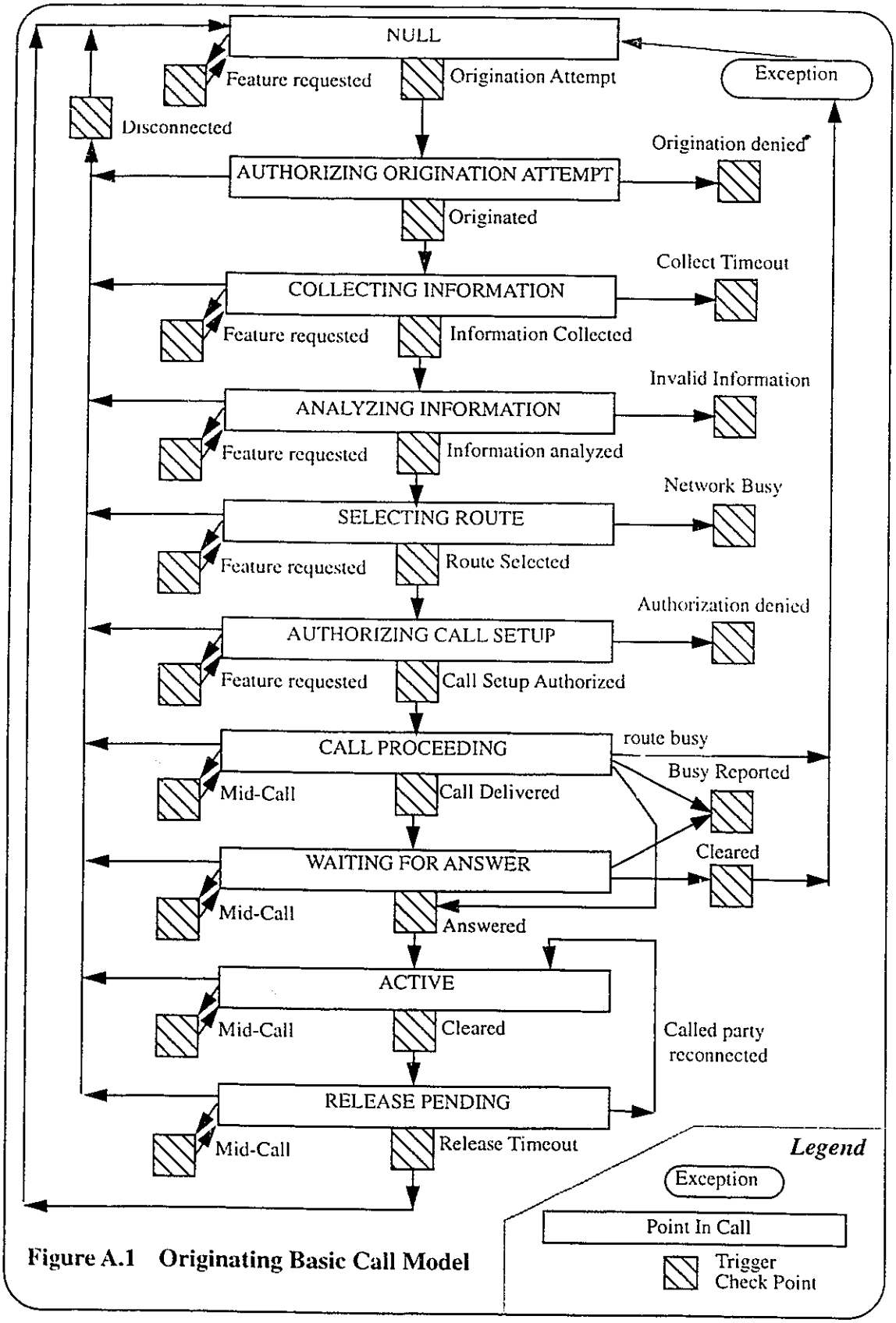


Figure A.1 Originating Basic Call Model

Action	No call exists. For a multi-party call with more than two parties, other calls may exist even if one call has returned to IDLE
Entry Event(s)	Disconnection and clearing of a previous call, after exception handling, or system (re)initialization
Exit Event(s)	• Origination Attempt event - an indication of desire to place an outgoing call. the switch receives an off-hook indication from an idle line.
Available Information	• Calling Party Automatic Number Identification (ANI) - the charge number of the calling party (information available in the switch). • Calling Party ID - the network address of the calling party (information available in the switch) • Calling Party Business Group ID - The Basic Business Group or multi-switch Business Group ID of the calling party (information available when the user is a member of a Business Group). • Customer Class of Service - code that is used to identify all attributes of a line that require distinctive call processing treatment • Bearer Capability - assumed to be 3.1 kHz audio or 56 kbps for non-ISDN users (for a line dedicated to Public Switched Digital Service)

Table A.1 Null PIC

Action	The switch verifies the authority of the user to place the call with the given properties (e.g. line restrictions) using the appropriate screening procedures. Any glare situation detected is resolved.
Entry Event(s)	Originating Attempt event
Exit Event(s)	• Originated Attempt event - detected when the authority to place an outgoing call is verified.
Exit Events related to Incomplete Calls	• Originated Denied event - detected when the authority to place an outgoing call is denied. • Disconnected event - Calling party disconnects and the switch receives an on-hook signal from a caller following switch hook flash timing.
Available Information	After detecting the Originated event, the switch is assumed to have the same information as in Table A.1. If the switch determines that the origination is denied, the cause of the failed authorization (e.g. Origination denied) is also known.

Table A.2 Authorizing Origination PIC

Action	The switch collects initial information (e.g. service codes, prefixes and address information) from the user. This information is collected according to a dialing template. Typically, the switch sends an in-band prompt to the user (dial tone) attaches a digit receiver collects digits and applies timing.
Entry Event(s)	Originated event
Exit Event(s)	• Information collected event - detected when the complete initial information from the caller is available. If overlap sending procedures are being used, this occurs when the first six digits have been collected. • Feature Requested event - detected when a switch hook flash is entered in the special situation where the user is initiating another two-party call in addition to an existing multi-party call
Exit Events related to Incomplete Calls	• Collect Timeout event - Complete initial information not received before a normal interdigit time expires • Collect Info failure - the switch is unable to collect the information because of a lack of resources (e.g. no available digit receivers) • Disconnect event.
Available Information	After the switch determines that information collection is complete, the switch is assumed to have the same information as in Table A.1 as well as the following Collected Information: • Access code (from a Business Group Dialing Plan; e.g. an access code to the public network, or a feature access code) • Star Prefix Service Code (two-digit code following a '*' or 11 prefix • Prefix (e.g. 1+, 0+, 01+, 011+, 0-, 00-) • Carrier Access Code • Collected Address information • Dialing plan of the address (could be International numbering plan, North American numbering plan, or a private numbering plan)

Table A.3 Collecting Information PIC

Action	The switc'. interprets and translates the collected information according to the specified numbering plan. The switch determines the: • called party ID, • type of call (e.g. intra-LATA or Inter-Exchange Carrier) and • Carrier. For calls to lines which are not subtending the switch, a route index is determined based on the information available (e.g. address) and the class of service of the line. This index is a pointer to a list of trunk groups. For calls received from an Inter-Exchange Carrier, the called address is screened to determine whether the first six digits of a ten-digit address or the first three digits of a seven-digit address subtend the switch.
Entry Event(s)	Information Collected event.
Exit Event(s)	Information Analyzed event - occurs when a called party ID, type of call and, when appropriate, a route index and a carrier are determined. It should be noted that the availability of a called party ID does not necessarily mean that the final address has been determined (e.g. multi-line hunt groups not searched yet or a Directory Number has not been translated into a physical line termination.
Exit Events related to Incomplete Calls	• Invalid Information event - collected information not valid • Disconnected event (see Table A.2 for definition.
Available Information	After the switch determines that information collection has been analyzed, the switch is assumed to have the same information as in Table A.1 as well as the following Analysis Results (of the collected information): • Called Party ID • Numbering Plan of the address • Type of Call which indicates the type of operator assistance required • Carrier • Carrier Selection which is an indication of whether the carrier was obtained from a user of from the network default carrier associated with the user • Originating Line information (see Table A.3) • Route Index/ Facility Restriction Level (FRL), which is an index to a route list of trunk groups (if the call does not terminate on this switch). If the route index points to a single trunk group and the trunk group is assigned an FRL, then the FRL is also known

Table A.4 Analyzing Information PIC

Action	To select the outgoing route, the switch interprets the analysis results. This may involve: • routing the call to a directory number on this switch, • searching an automatic flexible route list, • selecting a private facility trunk group, or • searching multiple route indices supplied by an external piece of equipment. For carrier calls, the circuit code is determined based on the user and the type of call. The individual destination resource of resource group (e.g. a specific line of a multi-line hunt group) is not selected. When the entry event is the route busy event from the Call Proceeding PIC (see Table A.7), depending on the location in the network where the route was busy, the switch will: • If the route is busy at this switch, the switch attempts to route the call via an alternate route. If there are no alternate routes or they have all been tried, the Network Busy event is detected • If the route is busy at another switch, an indication may be passed to this switch. A Network Busy event is detected. The switch may continue to collect information
Entry Event(s)	Information Analyzed event or Route Busy event reported from the Call Proceeding PIC (see Table A.7).
Exit Event(s)	• Route selected - occurs when the originating routing information for the call has been identified • Feature Requested event - (see Table A.7).
Exit Events related to Incomplete Calls	Network Busy event or Disconnected event (see Table A.2). The Network Busy event occurs when all the routes are busy.
Available Information	After the switch determines that the route has been selected, it is assumed that the following information is available: • identification information- see Table A.1, • analysis Results- see Table A.4. Also, additional route indices to route the call may be supplied from an external piece of equipment when the routes specified by the primary route index are busy. The switch remembers which routes and/or carriers have been tried for the call. • routing information- If the call is to an inter-exchange carrier, this will include the circuit code information. If this call is an AFR call, this contains the list of routes and an indication of which route to select next. Although the route index may point to a list of trunk groups to be tried for the call, only information about the current route being tried will be sent to the Authorizing Call Setup PIC (see Table A.6)

Table A.5 Selecting Route PIC

Action	The switch verifies the authority of the calling party to place this particular call (e.g. checking business group restrictions and toll restrictions). The switch may continue to collect information.
Entry Event(s)	Route Selected event
Exit Event(s)	• Call Setup Authorized (when the authority to place the call has been verified). • Feature Requested event - (see Table A.3)
Exit Events related to Incomplete Calls	• Authorization Denied event - occurs when authority to pace the call is not granted • Disconnected event (see Table A.3).
Available Information	After the switch determines that the route has been selected, it is assumed that the following information is available: • identification information- see Table A.1, • Analysis Results- see Table A.5. In this PIC, only the current route index and carrier being tried are known • Routing information- This may contain the circuit code (transit network selection parameter) information and a route (from an Automatic Flexible Route list) • Facility Restriction level (FRL)- This information may be available for the originating line or trunk and for the termination facility for calls to a private facility trunk group that use FRLs and is determined as part of the authorization procedures for this call.

Table A.6 Authorizing Call Setup PIC

Action	The switch sends an indication of desire to set up a call to the specified Called Party ID to the terminating call portion. The information which may be passed to the terminating call portion is: Calling party automatic number ID, Calling Party ID, Calling Party Business Group ID, carrier, transit network selection, Called Party ID, route index, and bearer capability. Other information not used by the processing modeled by this PIC may also be passed to the terminating call portion.
Entry Event(s)	Call Setup Authorized event.
Exit Event(s)	• Route busy - detected if an indication of a Busy event specifying route busy is received from the terminating call portion. In this case, the originating call portion returns to the Selecting Route PIC. • Answered - occurs when an indication of a Connected event is received from the terminating call portion. • Call Delivered - occurs when an indication of a Call Accepted event is received from the terminating call portion. • Feature Requested - corresponds to the user entering a switch hook flash.
Exit Events related to Incomplete Calls	• Busy reported event - occurs when an indication of a Busy event specifying user busy is received from the terminating portion of the call (i.e. network- determined-user-busy or user-determined-user-busy). This event also occurs when an indication of a Call Rejected event specifying user busy is received from the terminating portion of the call. If an indication of busy is received from the terminating portion of the call, busy tone is provided by the switch. • Cleared event - occurs when an indication of a Call Rejected event from the terminating call portion and this indication does not specify user busy (see Table A.14). When a Call Rejected indication is received, a tone or announcement may be provided from the switch. • Disconnected event - an indication of clearing is passed to the terminating call portion
Available Information	After the switch determines that the call has been delivered, it is assumed that the switch has the following information: • identification information (see Table A.1) • Analysis Results (see Table A.6) • Routing Information - this may contain the circuit code (as described by a transit network selection parameter) or a private facility trunk group

Table A.7 Call Proceeding PIC

Action	The switch waits for the terminating party to answer. At this point, the caller receives inband audible ringing (from the terminating switch). An indication of call progress may be received from the terminating call portion.
Entry Event(s)	Call Delivered event
Exit Event(s)	• Answered - occurs when an indication of a Connected event is received from the terminating portion of the call.Answered event occurs, the treatment applied is the same as that in (Table A.7) • Feature Requested (see Table A.7).
Exit Events related to Incomplete Calls	• Disconnected (see Table A.7) • Busy Reported (see Table A.7) • Cleared - occurs when an indication of Call Rejected not specifying user busy is received from the terminating call portion or when an indication of a Ringing timeout event is received from the terminating call portion (see Table A.15)
Available Information	When the switch is in this PIC, the switch has the following information: • identification information (see Table A.1) • Analysis Results (see Table A.5) • Routing Information - this may contain the circuit code (as described by a transit network selection parameter) or a private facility trunk group

Table A.8 Waiting for an Answer PIC

Action	Connection established between calling party and called party (see Table A.7). Call supervision is being provided
Entry Event(s)	• Answered event • Called Party Reconnected event (see Table A.10)
Exit Event(s)	• Disconnected event (see Table A.7) • Feature Requested event (see Table A.7) • Cleared event - occurs when an indication that the called party disconnected is received. For an intra-switch call, the switch receives an on-hook indication from the terminating line (and the switch has provided switch hook flash timing). The indication of Cleared passed to the originating call portion indicates timed release disconnect when timing is to be done and release when timing is not done. The Cleared event may also result from error situations.
Available Information	When the switch is in the ACTIVE PIC, the originating basic call model should have the following information: • Class of Service (see Table A.1) • Calling Party Automatic Number Identification • Calling Party ID • Calling Party Business Group ID • Bearer Capability (seeTable A.1) • Calling Party's Terminal Service Profile (see Table A.1)

Table A.9 Active PIC

Action	The connection is left intact and appropriate timing may be started, depending on the indication received, the access type and the position of the switch in the connection. If the terminating call portion sends an indication that the terminating party has gone off-hook, the ACTIVE PIC should be restored. If the originating party disconnects, the call is cleared without further delay.
Entry Event(s)	Cleared event indicating timed release disconnect or release
Exit Event(s)	• Feature Requested event (see Table A.7) • Disconnected event (see Table A.7) • Called Party Reconnected event - occurs when the terminating call portion sends an indication that the terminating party has gone off-hook. • Release Timeout event - occurs when the timed release disconnect timer expires (at this switch).
Available Information	The information available in this PIC is the same as in Table A.9.

Table A.10 Release Pending PIC

A.2 Terminating Basic Call Model

The Terminating Basic Call Model (Fig. A.2) covers three basic stages of Call Processing (corresponding PICs given for each):

- Call setup: NULL, AUTHORIZING TERMINATION, HUNTING FACILITY, PRESENTING CALL
- Stable Call: ALERTING, ACTIVE
- Call Clearing: RELEASE PENDING

The following tables give detailed descriptions of each of the PICs in (Fig. A.2) using the same table format as was used earlier.

Action	The line is idle (i.e. no call exists) and the switch provides supervision on the line.
Entry Event(s)	Disconnection and clearing of a previous call, after exception handling, or system (re)initialization.
Exit Event(s)	• Termination Attempt - occurs when an indication of a desire to deliver a call is received from the originating portion of the call.
Available Information	• Calling Party Automatic Number Identification (ANI) - the charge number of the calling party (information available in the switch). • Calling Party ID - the network address of the calling party (information available in the switch) • Calling Party Business Group ID - The Basic Business Group or multi-switch Business Group ID of the calling party (information available when the user is a member of a Business Group). • Carrier - indicates whether a call should be routed over BellCore Client Company facilities, a private network/facility or to a specific Inter-exchange Carrier/ International Carrier. It is passed from the originating portion of the call • Circuit Code - information from the originating portion of the call about which of the Inter-exchange/ International carrier facilities to select for routing the call. • Originating Line Information - information from the originating portion of the call to do with the class of service of the originating user. • Bearer Capability - information passed from the originating portion of the call.

Table A.11 Null PIC

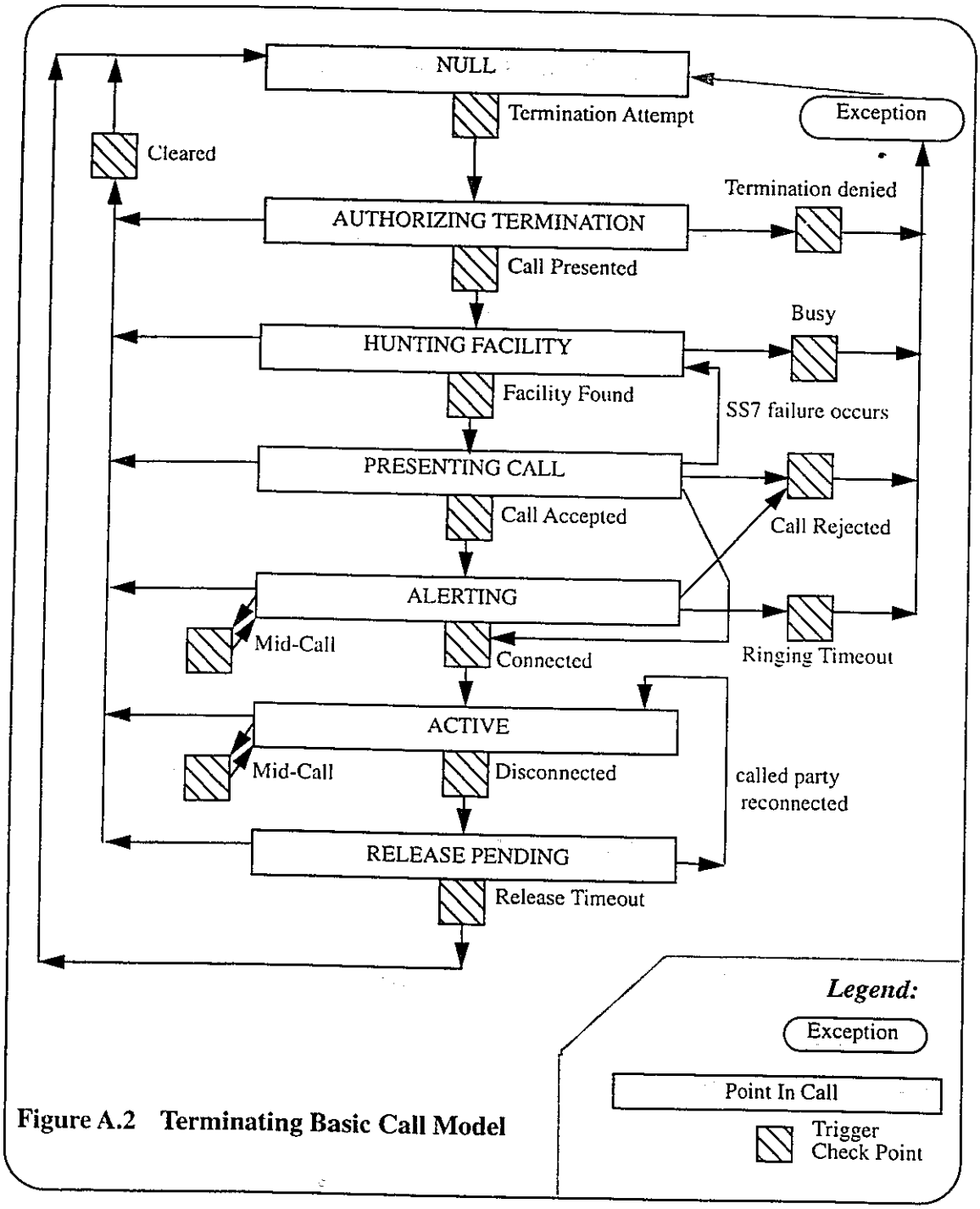


Figure A.2 Terminating Basic Call Model

Action	The switch verifies the authority to route this call to the terminating access (e.g. check business group restrictions, restricted incoming access to line, or bearer capability compatibility).
Entry Event(s)	Termination Attempt event
Exit Event(s)	Call Presented event
Exit Events related to Incomplete Calls	• Termination Denied event - occurs when the authority to route this call to the terminating user is denied. • Cleared event - occurs when an indication of clearing is received from the origination portion of the call.
Available Information	It is assumed that the switch has the same information available for the terminating call portion after the Call Presented event is detected as it does when the Termination Attempt event is detected.

Table A.12 Authorizing Termination PIC

Action	The busy/idle status of the terminating access is determined. If the line is already involved with an existing call, the line is treated as network-determined user busy.
Entry Event(s)	Call Presented event or a re-attempt in Presenting Call.
Exit Event(s)	• Facility Found event - occurs when the terminating access is not busy (i.e. an idle facility could be found or a position in a queue is available).
Exit Events related to Incomplete Calls	• Busy event - occurs when the terminating access is busy (no idle facility can be found or no position in a queue is available). the Busy event may also be detected as a result of an analog line being out of order, marked as busy by a customer make-busy key, or as a result of certain maintenance actions. Note that when Busy is passed to the originating call portion, the terminating Basic call Model goes away. If a terminating feature acts on the Busy event and changes the event (e.g. Call Waiting feature, the event will not get passed to the Originating Basic call Model. • Cleared event (see Table A.12)
Available Information	When the Facility Found event is detected, it is assumed that the switch has the following available information: • identification information (see Table A.11) • Facility Group - For calls routed out of this switch, this identifies the Trunk Group (private or public) that has been selected to route call on. This may also identify the Multi-Line Hunt Group for lines terminating on the switch. • Facility Group Member - For calls routed out of this switch, this identifies the Trunk Group (private or public) that has been selected to route call on. This may also identify the hunt-terminal within the Multi-Line Hunt Group for lines terminating on the switch.

Table A.13 Hunting Facility PIC

Action	The switch informs the terminating access of the call. The switch seizes the terminating line and applies power ringing. The line may or may not be part of a Multi-Line-Hunt-Group (or MLHG-I)
Entry Event(s)	Facility Found event
Exit Event(s)	• Call Accepted event - the switch provides power ringing to the terminating line. An indication of the event is passed to the originating portion of the call. • Connected event - occurs when the switch receives an indication that the terminating party has gone off-hook.
Exit Events related to Incomplete Calls	• Call Rejected event - occurs when an expected response is not received. An indication of the event is passed to the originating portion of the call. • Call Cleared event.
Available Information	• identification information (see Table A.11) • Facility Group information (see Table A.13) • Information regarding the call connection - e.g. a statement that the call type is non-ISDN.

Table A.14 Presenting Call PIC

Action	The switch alerts the terminating resource and waits for the terminating party to answer. the ringing timer is started if the terminating party subscribes to the timer. A call progress indication and any received information is passed to the originating call portion.
Entry Event(s)	Call Accepted event.
Exit Event(s)	• Feature Requested event (see Table A.7) • Connected event - occurs when the call is answered by the terminating party. An indication of Connected event is passed to the originating portion of the call.
Exit Events related to Incomplete Calls	• Ringing Timeout event - detected when the terminating party does not answer before the switch-based ringing timer expires • Call Rejected event - passed to the originating call portion • Cleared event - passed to the originating call portion
Available Information	It is assumed that the switch has the same information available as it had after the "Call Accepted" event was detected. Updated information on the call may be received in the Call Progress Message.

Table A.15 Alerting PIC

Action	Connection established between calling and terminating party. Call supervision is provided.
Entry Event(s)	• Connected event • Called Party Reconnected event (see Table A.17)
Exit Event(s)	• Feature Requested event (see Table A.7) • Disconnected event - this results from the switch receiving an on-hook indication • Cleared event
Available Information	• Bearer Capability (see Table A.11) • Called Party ID • Called Party Business Group ID • Facility Group (see Table A.13) • Facility Group member (see Table A.13)

Table A.16 Active PIC

Action	The connection is left intact and the appropriate timing may be started, depending on the access type and the position of the switch in the connection. If the terminating party goes off-hook before the timer expires, the ACTIVE PIC is restored and an indication of this is passed to the originating call portion. If the originating party disconnects, the call is cleared without further delay.
Entry Event(s)	Disconnected event
Exit Event(s)	• Release Timeout event - occurs when the release timer expires (at this switch). An indication of the Release Timeout event is passed to the originating call portion. • Called Party Reconnected event - occurs when the terminating party goes off-hook. An indication of the Called Party Reconnected event is passed to the originating call portion. • Cleared event
Available Information	It is assumed that the switch has the same information as shown in Table A.16.

Table A.17 Release Pending PIC

Appendix B

Verilog Code

This Appendix contains the code used for the experiments described in earlier sections of the document (see section 5.2 on page 38). The hierarchy for the tasks, functions and modules in the simulation is shown with text indentation in (Fig. B.1).

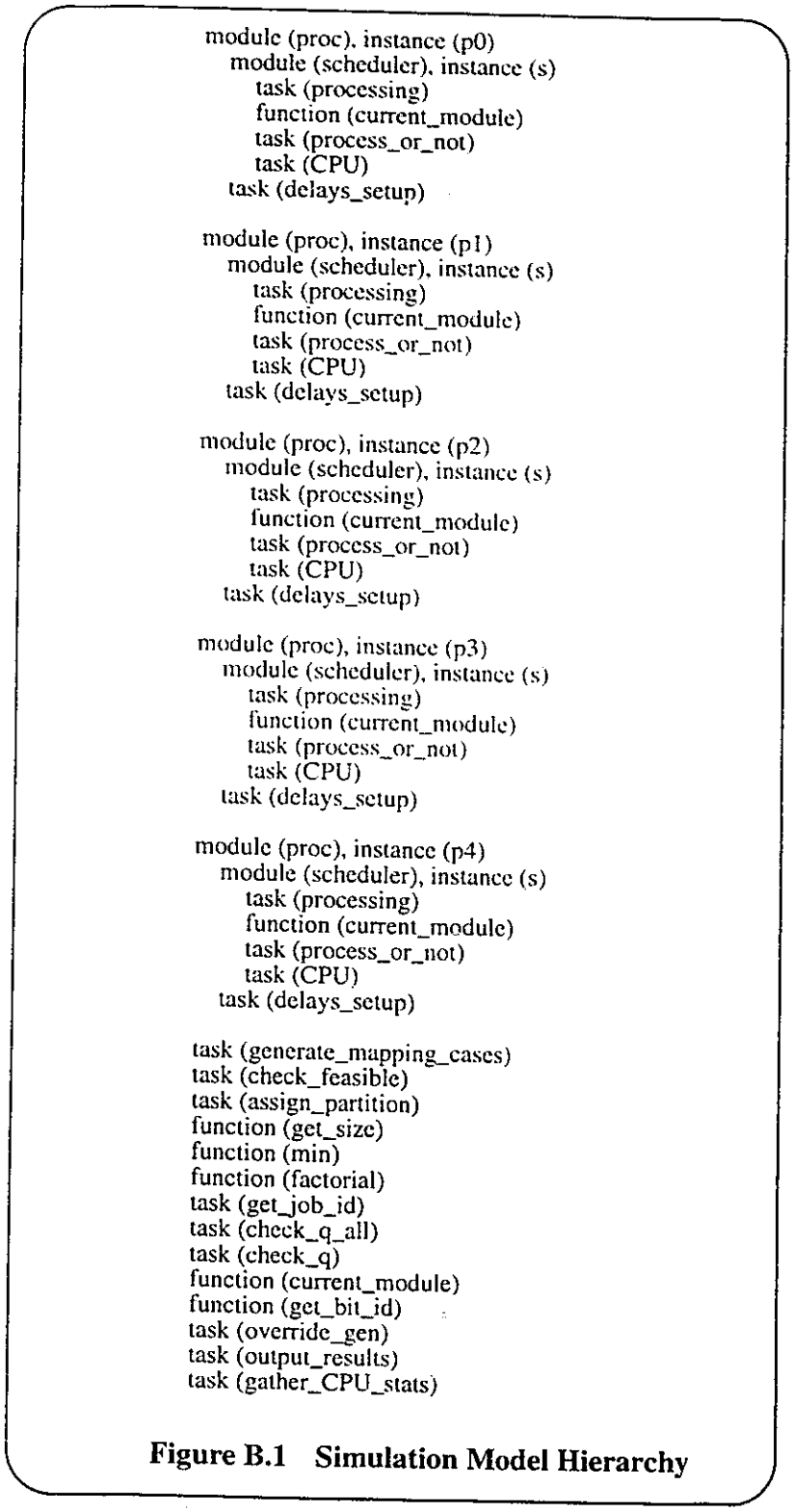


Figure B.1 Simulation Model Hierarchy

B.1 top.v

```
/* top.v */
`include "defs.v"

module top();

`include "vars.v"

proc #0 p0 (ready_1to0,id_in,sig_in,id_0to1,sig_0to1,sys_ready);
proc #1 p1 (ready_2to1,id_0to1,sig_0to1,id_1to2,sig_1to2,ready_1to0);
proc #2 p2 (ready_3to2,id_1to2,sig_1to2,id_2to3,sig_2to3,ready_2to1);
proc #3 p3 (ready_4to3,id_2to3,sig_2to3,id_3to4,sig_3to4,ready_3to2);
proc #4 p4 (ready_last,id_3to4,sig_3to4,id_out,sig_out,ready_4to3);

`include "setup.v"

initial
    begin
    epoch_count=0;
    last_total_jobs=0;
    for (i=0; i<= `p-1; i=i+1)
        begin
        busy_count[i]=0;
        last_busy_time[i]=$time;
        end
    end

always #`clk_period_div2
    begin
    clk = ~clk;
    epoch_count=epoch_count+1;
    if (epoch_count==`epoch_width)
        begin
        epoch_count=0;
        -> idle_epoch;
        end
    end

always @(idle_epoch)
    begin
    gather_CPU_stats;
    if (total_jobs<`total_jobs) output_results(`TRUE);
    for (i=0; i<= `p-1; i=i+1)
        begin
        busy_count[i]=0;
        last_busy_time[i]=$time;
        end
    last_total_jobs=total_jobs;
    end

always @(negedge p0.s.CPUhold) busy_count[0]=busy_count[0]+$time-last_busy_time[0];
always @(posedge p0.s.CPUhold) last_busy_time[0]=$time;
always @(negedge p1.s.CPUhold) busy_count[1]=busy_count[1]+$time-last_busy_time[1];
always @(posedge p1.s.CPUhold) last_busy_time[1]=$time;
always @(negedge p2.s.CPUhold) busy_count[2]=busy_count[2]+$time-last_busy_time[2];
always @(posedge p2.s.CPUhold) last_busy_time[2]=$time;
always @(negedge p3.s.CPUhold) busy_count[3]=busy_count[3]+$time-last_busy_time[3];
always @(posedge p3.s.CPUhold) last_busy_time[3]=$time;
always @(negedge p4.s.CPUhold) busy_count[4]=busy_count[4]+$time-last_busy_time[4];
always @(posedge p4.s.CPUhold) last_busy_time[4]=$time;

`include "generate.v"
```

```
`include "statistics.v"

`include "override_gen_task.v"

`include "output_results_task.v"

`include "gather_CPU_stats_task.v"

endmodule // top

`include "proc.v"
```

B.2 defs.v

```
/* defs.v */
`timescale 1ms/1us
`define searchid 99999
`define silent `TRUE
// define silent `FALSE
`define total_jobs 1000
`define total_runs 127

/* Global definitions for various simulation parameters */
`define p 8                                  // number of processors in the pipeline
`define p_bits 4                             // lg base2 of above
`define m 10                                 // number of modules in the program
`define m_bits 4                             // lg base2 of above.
`define max_codes 10000
`define infinity 999999999
// `define pfactorial 120
`define pfactorial 40320
`define lg2pfactorial 16

`define mixinc 10
`define mixsum 120
`define lg2inc 8

/*
`define j 31       // number of jobs in the simulation
`define j_bits 5// lg base2 of above
*/
`define j 2047    // number of jobs in the simulation
`define j_bits 11// lg base2 of above
`define delay_bits 8// number of bits to store delay bits
`define user_delay_bits 12 // number of bits to store max user delay
`define user_vector `m'b0000100010// boolean to force exit from pipeline
`define user_seed 23
`define proc_delay_bits 12 // number of bits to store max processor delay
`define rx_q_size 150 // max number of entries in incoming message queue
`define rx_q_threshold 30          // maximum tolerated number of messages in incoming message queue
`define ready_q_size 100           // max number of entries in message queue of jobs from user
`define TRUE 1'b1
`define FALSE 1'b0
`define forwards 1'b1
`define backwards 1'b0
`define sigdec 1000000000
`define FIFO 1
`define LIFO 2
`define OK 0
`define q_empty 3
`define q_length 1
`define idle_count_threshold 20// number of time units to mark an 'idle' epoch
`define epoch_width `clk_period_div2*2// duration of time for calculating idle time
// `define epoch_width 100// duration of time for calculating idle time
`define dialtone_mean 3000
`define max_retry 32000
`define traffic_seed 27
// `define traffic_mean 2000
`define traffic_mean 100
`define user_q_id `p+4
`define output_q_id `p+5
`define job_q_id `p+6
`define seq_q_id `p+7
`define override_q_id `p+8
`define working_q_id (`override_q_id+1)
```

```
`define working2_q_id (`override_q_id+2)
`define top100_q_id (`override_q_id+3)
`define max_top100_q_size 100
`define mix_assign_q_id (`override_q_id+4)
`define mix_assign_q_size 1000
// insert other queues here, since there are multiple ids for the assign ones,
// which start at 100 and go on to 100+m
`define assign_q_id (`override_q_id+100)
`define jobmix_q_id (`assign_q_id+100)
`define jobmix_q_size 50
`define clk_period_div2 100
//`define compress_delay 40
`define compress_delay 5
//`define decompress_delay 40
`define decompress_delay 5
//`define tx_msg_delay 40
`define tx_msg_delay 5
//`define rx_msg_delay 40
`define rx_msg_delay 5
//`define retrieve_delay 40
`define retrieve_delay 5
//`define ISR_passthrough 20
`define ISR_passthrough 5
//`define scheduler_passthrough 40
`define scheduler_passthrough 5
`define max_delay 32766
`define inter_proc_OH 1
```

B.3 vars.v

```
/* vars.v */

wire [`j_bits-1:0]id_0to1;
wire [`j_bits-1:0]id_1to2;
wire [`j_bits-1:0]id_2to3;
wire [`j_bits-1:0]id_3to4;
wire [`j_bits-1:0]id_4to5;
wire [`j_bits-1:0]id_5to6;
wire [`j_bits-1:0]id_6to7;
wire [`j_bits-1:0]id_7to8;
wire [`j_bits-1:0]id_out;

wire sig_0to1, sig_1to2, sig_2to3, sig_3to4, sig_4to5, sig_out;
wire sig_5to6, sig_6to7, sig_7to8;

wire ready_1to0, ready_2to1, ready_3to2, ready_4to3, ready_5to4;
wire ready_6to5, ready_7to6, ready_8to7;
wire sys_ready;

reg ready_last;

reg [`j_bits-1:0]latched_id_out;

reg clk;
reg clkstat;
reg clk2;
reg [`m-1:0]temp;
integer i, j;

reg [`p_bits+3:0]tempvar1;

time temp2, temp3;
time last_time;
integer temp1, stat_val, stat_code, stat_stat;
reg [`m_bits:0]prog_module;

event traffic_generate1, traffic_generate2;
event idle_epoch;
integer seed_traffic;
integer next_packet;

integer mean, seed, seed1;
integer delay, seq_status, job_status, user_status, user_status2, job_id;
integer CPU_seed;
integer j_var, j_var1, j_var2, j_var3, u_var, user_id_out;
reg[`j_bits:0]job_var;
integer gen_status;

reg [`j_bits-1:0]id_in;
reg sig_in;
reg [`j_bits-1:0]jobdex;

reg [`j-1:0]j_q_id;
reg [`j_bits:0]j_qdex;
reg [`j-1:0]s_q_id;
reg [`j-1:0]o_q_id;
reg [`j-1:0]u_q_id;
reg [(`p*`j)-1:0]system_q_id;

reg [63:0]birth_time[0:`j-1];// creation time of job
reg [`m-1:0]processing_state[0:`j-1];
```

```
// tracking of which modules have been procesed for each job;
// bit set if already processed
reg [`m-1:0]override[0:`p-1];// override booleans for each proc.
integer override_seed;
reg [(`m_bits*`p)-1:0] override_id;
reg override_repeat;
reg [`m-1:0]user_ip_reqd;  // user input booleans for each module.
reg [`j:0]wakeup;          // bit set in bit position corresponding to job id
                           // if job_id is being activated into pipeline again
reg [`j_bits-1:0]twakeup;
reg kick_me;
reg [`j-1:0]passthrough;        // bit set if job in corresponding bit position is
                                // not to be processed further in the array
reg [`j_bits-1:0]round_robin;   // round_robin scheduling of new jobs (seed)
                                // bit set if a module must be processed
reg [`user_delay_bits-1:0]user_delay_ave[0:`m-1];// different statistics for each module

integer all_delay_specific [0:(`p*`m)-1]; // processor-specific module delays
integer temp_all_delay_specific [0:(`p*`m)-1]; // processor-specific module delays
integer all_orderings_delay_specific [0:(`p*`m*`pfactorial)-1]; // processor-specific module delays

reg [31:0]user_delay_seed[0:`m-1];                  // different statistics for each module
reg [(`proc_delay_bits-1+`p_bits):0]proc_delay[0:`j-1]; // cumulative processing delay (only)
reg [(`user_delay_bits-1+`m_bits):0]user_delay[0:`j-1]; // cumulative user delay (only)

time busy_count[`p-1:0];
time last_busy_time[`p-1:0];
integer epoch_count;
integer user_retry;
integer dialtone_seed;
integer uniform_seed;
time generated_jobs, total_jobs, total_runs, total_OH, total_proc, total_user;
time last_time_out;
time first_job_time;
time last_total_jobs;
integer procdex;
integer t2, t3, t4, t6;

reg [(`m_bits*(`m+1))-1:0]codes[0:`max_codes];
integer codesdex;
integer minimum;
reg [(`m_bits-1):0] index [0:`m];
time globalvar;

time bestdelay;
time worstdelay;
```

B.4 setup.v

```
/* setup.v */
initial
      begin
      jobdex=0; job_var=0; job_status=0; job_id=0;
      codesdex=0; globalvar=0;
      seed_traffic=`traffic_seed;
      wakeup=0;
      kick_me=0;
      round_robin=0;
      uniform_seed=1;
      override_seed=3;
      dialtone_seed=3456;
      CPU_seed=23;
      user_retry=0; total_OH=0; total_proc=0; total_user=0;
      generated_jobs=0; total_jobs=0; total_runs=0;
      user_ip_reqd=`user_vector;
      ready_last=`TRUE;
      output_results(`FALSE);

      $q_initialize(`job_q_id,`FIFO,`j+1,job_status);
      $q_initialize(`seq_q_id,`FIFO,`j+1,job_status);
      $q_initialize(`output_q_id,`FIFO,`j+1,job_status);
      $q_initialize(`user_q_id,`FIFO,`j+1,job_status);
      $q_initialize(`override_q_id,`FIFO,`total_runs+1,job_status);
      $q_initialize(`mix_assign_q_id,`FIFO,`mix_assign_q_size,job_status);

      for (i=0; i<=`m-1; i=i+1)
            begin
            $q_initialize(`assign_q_id+i,`FIFO,`pfactorial,job_status);
            end

      $q_initialize(`working_q_id,`FIFO,`p+1,job_status);
      $q_initialize(`working2_q_id,`FIFO,`p+1,job_status);
      $q_initialize(`top100_q_id,`LIFO,100,job_status);

      bestdelay=`infinity;
      worstdelay=`infinity;

      clk=0;
      clk2=0;
      override_gen(override_id);
      $readmemh("fuser_delay_ave",user_delay_ave);
      for (i=0; i<=`m-1; i=i+1)
            begin
            user_delay_seed[i]=`user_seed+i;
            end
      $display ("Input files read");
      $display("setup.v:DATAMEANSVARS user_ip_reqd=%b",user_ip_reqd);
      generate_mapping_cases;

//    setup_delay_combinations;
      end

// `include "setup_schedules_task.v"

`include "generate_mapping_cases_task.v"
// `include "setup_delay_combinations_task.v"
```

B.5 generate_mapping_cases task

```
task generate_mapping_cases;

reg [`m_bits-1:0]ending[0:`m+1];
reg [`m_bits-1:0]index[0:`m];
integer i, k, count, last_i, minimum;
integer q_size,temp,temp1,dex;
reg [(`m*`p)-1:0] assignment;
reg [(`m*`lg2inc)-1:0] job_ratio;
reg new_partition;
reg [`m_bits-1:0]num_partitions;
reg [`m_bits-1:0]partition_size;
reg [`m_bits-1:0]size_count;
reg [`m-1:0]user_vector_copy;
reg [`m_bits-1:0]part_size[0:`m-1];
reg [`m_bits-1:0]part_end[0:`m-1];
time david;

  begin // task

  user_vector_copy=`user_vector;
  num_partitions=0;
  partition_size=get_size(user_vector_copy,0,`forwards);
  new_partition=`TRUE;

  // reset to min(`m,`p) at start of each partition
  // and then partition_size/2, partition_size/3,....
  // until next partition reached

  for (i=0;i<`m;i=i+1)
    begin
    if(user_vector_copy[i]==1 || i==(`m-1))
      begin
      if (new_partition==`TRUE) ending[i]=min(partition_size,`p); // previous digit was a one
      else ending[i]=partition_size/size_count; // previous digit was a zero
      partition_size=get_size(user_vector_copy,i,`backwards);
      part_size[num_partitions]=partition_size;
      part_end[num_partitions]=i;
      num_partitions=num_partitions+1;
      new_partition=`TRUE; // indicate that this digit is 1 for next time
      end
    else
      begin
      if (new_partition==`TRUE) // first zero after a one
        begin
        partition_size=get_size(user_vector_copy,i,`forwards);
        ending[i]=min(partition_size,`p);
        size_count=2;
        new_partition=`FALSE;
        end
      else // zero after a zero
        begin
        ending[i]=partition_size/size_count;
        size_count=size_count+1;
        end
      end
    end

  david=0;

for (index[0]=0; index[0]<=ending[0]; index[0]=index[0]+1)
 begin
```

```
      for (index[1]=0; index[1]<=ending[1]; index[1]=index[1]+1)
       begin
       for (index[2]=0; index[2]<=ending[2]; index[2]=index[2]+1)
        begin
        for (index[3]=0; index[3]<=ending[3]; index[3]=index[3]+1)
         begin
         for (index[4]=0; index[4]<=ending[4]; index[4]=index[4]+1)
          begin
          for (index[5]=0; index[5]<=ending[5]; index[5]=index[5]+1)
           begin
           for (index[6]=0; index[6]<=ending[6]; index[6]=index[6]+1)
            begin
            for (index[7]=0; index[7]<=ending[7]; index[7]=index[7]+1)
             begin
             for (index[8]=0; index[8]<=ending[8]; index[8]=index[8]+1)
              begin
              for (index[`m-1]=0; index[`m-1]<=ending[`m-1]; index[`m-1]=index[`m-1]+1)
               begin
                david=david+1;
//               if (david>= 3660) $stop;
               if (david%10==0) $display("+ david=%0d",david);
               if (david>3660) $display("+ david=%0d",david);
              check_feasible(num_partitions);
              end // m
             end // 9
            end // 8
           end // 7
          end // 6
         end // 5
        end // 4
       end // 3
      end // 2
     end // 1

     setup_delay_combinations (num_partitions);
//   combinations_task(num_partitions);

    $q_exam(`mix_assign_q_id,`q_length,q_size,temp);
    $display("%0d job ratios produced",q_size);
    for (i=0; i<q_size;i=i+1)
       begin
       $q_remove(`mix_assign_q_id,dex,temp,temp1);
       $q_add(`mix_assign_q_id,dex,temp,temp1);
       job_ratio=calculate_cost.mix_ratio[dex];
       assignment=calculate_cost.mix_assignment[dex];
       $write("RATIO ");
       for (j=0; j<num_partitions;j=j+1)
            begin
            $write("%0d  ",job_ratio[`lg2inc-1:0] );
            job_ratio=job_ratio>>`lg2inc;
            end
       $display("");
       end

   end // task

endtask
```

B.6 check_feasible task

```
task check_feasible;
// Check combinations of assignment counts of consecutive modules in partition
// which are assigned to processors meet the following constraints:
//   - more module 'runs' than number of processors is not possible
//   - modules that are in a partition must all be assigned
input [`m_bits-1:0]num_partitions;

reg var;
reg [`p_bits-1:0]nonzero_count[0:`p-1];
reg [(`m_bits*(`m+1))-1:0] value[0:`m-1]; // m partitions of m+1 integer counts (each m_bits max)
integer nonzero;
reg [(`m_bits*(`m+1))-1:0] temp; //
integer i,j,k,sum,mod_dex;
reg [`m_bits-1:0]local_index;
reg [`m_bits-1:0]local_part_size;
reg feasible;
reg terminate;

  begin // task


  j=0; mod_dex=0;
  feasible=`TRUE;

  while ( (j<num_partitions) && (feasible==`TRUE) ) // for each partition
    begin
    sum=0; i=0; nonzero=0; terminate=`FALSE;
    local_part_size=generate_mapping_cases.part_size[j];
    while ((i<local_part_size) && terminate==`FALSE)
      begin
      local_index=generate_mapping_cases.index[mod_dex];
      sum=sum+((i+1)*local_index);
      // local_index gives the number of occurrences of the number i+1 (i.e. index[4]=2 means there are two 5's)
      if (local_index!=0) nonzero=nonzero+local_index;
      if (sum>local_part_size || nonzero>`p)
      // sum exceeds number of modules in partition or more modules than processors
        begin
        sum=0;
        terminate=`TRUE;
//      $display("check_feasible: value is impossible");
        end
      else
        begin
        i=i+1;
        mod_dex=mod_dex+1;
        end
      end // while i
    if (sum==local_part_size)
    // feasible if total number of digits is equal to the number of modules in the partition
      begin
      feasible=`TRUE;
//    $display("check_feasible: value still feasible");
      nonzero_count[j]=nonzero;
      j=j+1;
      end
    else
      begin
//    $display("check_feasible: value not feasible");
      feasible=`FALSE;
      end
    end // while j
```

```
    if (feasible==`TRUE)
      begin
//    $display("check_feasible: value feasible; proceeding with next check");
      mod_dex=0;
      for (i=0; i<num_partitions; i=i+1)
        begin
        value[i]= `p-nonzero_count[i]; // number of unassigned processors
        for (j=0; j<generate_mapping_cases.part_size[i]; j=j+1)
          begin
        // compile indices for partition (giving number of occurrences of digit j)
          temp=generate_mapping_cases.index[mod_dex]<<((j+1)*`m_bits);
          value[i]=value[i] | temp;
          mod_dex=mod_dex+1;
          end
        end

        assign_partition(num_partitions);
      end
    end //task

endtask
```

B.7 assign_partition task

```
task assign_partition;
input [`m_bits-1:0]num_partitions;

reg [(`p*`m_bits)-1:0]permutation[0:(`pfactorial*`m)-1];
reg [(`p*`m_bits)-1:0]temp_perm;
reg [(`p*`m_bits)-1:0]result;
// maximum of m partition assignments to p processors
// (m_bit values) and up to p factorial orderings for each partition
reg [((`m+1)*`m_bits)-1:0]value_temp;
reg [`m_bits-1:0]temp;
reg [`m_bits-1:0]local_value; // single integer count (m_bits max)
reg [`lg2pfactorial-1:0]part_q_size[0:`m-1];
integer digit, q_size, tempdex;
integer i,j,k,dex,partition,try;
integer temp1,temp2;
reg already_there;

  begin // task

  for (partition=0; partition<num_partitions; partition=partition+1)
   // for every partition, generate all possible arrangements of p processors (p factorial of them)
   begin
   for (index[0]=0; index[0]<`p; index[0]=index[0]+1)
    begin
    for (index[1]=0; index[1]<`p-1; index[1]=index[1]+1)
     begin
     for (index[2]=0; index[2]<`p-2; index[2]=index[2]+1)
      begin
      for (index[3]=0; index[3]<`p-3; index[3]=index[3]+1)
       begin
       for (index[4]=0; index[4]<`p-4; index[4]=index[4]+1)
         begin // set up global assignment table for partition
for (index[5]=0; index[5]<`p-5; index[5]=index[5]+1)
 begin // set up global assignment table for partition
 for (index[6]=0; index[6]<`p-6; index[6]=index[6]+1)
  begin // set up global assignment table for partition
  for (index[7]=0; index[7]<`p-7; index[7]=index[7]+1)
   begin // set up global assignment table for partition
        for (i=0; i<=generate_mapping_cases.part_size[partition];i=i+1)
          // for numbers of program modules in partition
          // largest non-zero integer count is less than the partition size
           begin
           value_temp=check_feasible.value[partition];
           value_temp=value_temp>>(`m_bits*i);
           local_value=value_temp[`m_bits-1:0];
           // (m+1) m_bits values for each assignment of modules in the partition to p processors
        Sq_exam(`working_q_id,`q_length,q_size,temp1);
        if (q_size!=0 && i==0) $display("Error: q_size =%0d and not empty",q_size);
           while (local_value > 0)
           // add appropriate quantities of digit i to queue
             begin
             $q_add(`working_q_id,i,temp1,temp2);
//           $display("assign partition: added %0d to working queue (local_value=%0d)",i,local_value);
             local_value=local_value-1;
             end
          end // for i
        // now have queue which contains the numbers of consecutive modules in the partition to be
        result=0;
        Sq_exam(`working_q_id,`q_length,q_size,temp1);
        if (q_size != `p) $display("Error: q_size =%0d and not equal to p",q_size);
        // assigned over p processors; note that the sum of the p items in the queue is equal to the partition size
```

```
        for (digit=0; digit<`p; digit=digit+1)
          // generate appropriate ordering of p digits based on index
          begin
          $q_exam(`working_q_id,`q_length,q_size,temp1);
          for (dex=0; dex<q_size;dex=dex+1)
            begin
            $q_remove(`working_q_id,try,temp1,temp2);
//            $display("assign_partition: removed item %0d from queue (maybe to be put back)",try);
            if (dex != index[digit])
              // index gives the number of the item in the queue
              begin
              // not selected; put back to end of queue to retain relative order
              $q_add(`working_q_id,try,temp1,temp2);
              end
            else
              begin
              // selected; concatenate to result
              result=(result<<`m_bits) | try[`m_bits-1:0]; // compile register of results
              end
            end // for dex loop
          $q_exam(`working_q_id,`q_length,q_size,temp1);
          end // for digit loop
          $q_exam(`working_q_id,`q_length,q_size,temp1);

        already_there=`FALSE; // initialize search termination variable
        dex=1;
        $q_exam(`assign_q_id+partition,`q_length,q_size,temp1);
        while (dex<=q_size && already_there==`FALSE) // check if item in main queue already
          begin
          $q_remove(`assign_q_id+partition,tempdex,temp1,temp2);
          if (permutation[(`pfactorial*partition)+tempdex] == result)
              begin
              already_there=`TRUE;
              end
          $q_add(`assign_q_id+partition,tempdex,temp1,temp2); // add old item back to queue, regardless
          dex=dex+1;
          end // while dex loop
        if (dex==(q_size+1) && already_there==`FALSE)
          begin
          // if item is not in the queue already and the whole queue has been searched, add it to the queue
//          $display("assign_partition: >>> adding item id=%0d (value=%h) to assign queue %0d",q_size,result,partition);
          $q_exam(`assign_q_id+partition,`q_length,q_size,temp1);
          $q_add(`assign_q_id+partition,q_size,temp1,temp2);
          permutation[(`pfactorial*partition)+q_size] = result;
          end // if (dex...
end // 7
end // 6
end // 5
        end // 4
      end // 3
     end // 2
    end // 1
   end // 0
  $q_exam(`assign_q_id+partition,`q_length,q_size,temp1);
  part_q_size[partition]=q_size;

  // only need lg2(p!) bits to store a maximum queue size variable of p!
  end // partition

 end //task
endtask
```

B.8 setup_delay_cominations task

```
task setup_delay_combinations;
input [`m_bits-1:0]num_partitions;

integer i,j,k;
integer procdex,try,count, q_size;
reg [(`p_bits*`p)-1:0]proc_ordering[0:`max_top100_q_size-1];
reg [(`p_bits*`p)-1:0]temp_ordering;
reg [`p_bits-1:0]p_order[0:`p-1];
integer order [0:`p-1];
// reg [(`m*`p)-1:0] assignment_last;

begin // task
 $readmemh("fdelay_specific",top.temp_all_delay_specific);
 count=0;
 for (order[0]=0; order[0]<1; order[0]=order[0]+1)
  begin
  for (order[1]=0; order[1]<1; order[1]=order[1]+1)
   begin
   for (order[2]=0; order[2]<1; order[2]=order[2]+1)
    begin
    for (order[3]=0; order[3]<1; order[3]=order[3]+1)
     begin
     for (order[4]=0; order[4]<1; order[4]=order[4]+1)
       begin
       for (i=0; i<`p; i=i+1)
            begin
            $q_add(`working2_q_id,i,temp1,temp2);
            end
//     $display("");
       for (procdex=0; procdex<`p; procdex=procdex+1)
            begin
            $q_exam(`working2_q_id,`q_length,q_size,temp1);
            for (j=0; j<q_size; j=j+1)
                 begin
                 $q_remove(`working2_q_id,try,temp1,temp2);
                 if (j != order[procdex])
                      begin
                      $q_add(`working2_q_id,try,temp1,temp2);
                      end
                 else
                      begin
                      for (k=0; k<`m; k=k+1)
                           begin
                           top.all_delay_specific[procdex*`m+k]=top.temp_all_delay_specific[try*`m+k];
                           top.all_orderings_delay_specific[(count*`p*`m)+(procdex*`m)+k]=top.temp_all_delay_specific[try*`m+k];
                           end // for k
                      end
                 end // for j
            end // for procdex

$display("");
$display("%0h %0h %0h %0h %0h %0h %0h %0h %0h %0h",
top.all_delay_specific[0*`m+0],
top.all_delay_specific[0*`m+1],
top.all_delay_specific[0*`m+2],
top.all_delay_specific[0*`m+3],
top.all_delay_specific[0*`m+4],
top.all_delay_specific[0*`m+5],
top.all_delay_specific[0*`m+6],
top.all_delay_specific[0*`m+7],
top.all_delay_specific[0*`m+8],
```

```
top.all_delay_specific[0*`m+9]);
$display("%0h %0h %0h %0h %0h %0h %0h %0h %0h %0h",
top.all_delay_specific[1*`m+0],
top.all_delay_specific[1*`m+1],
top.all_delay_specific[1*`m+2],
top.all_delay_specific[1*`m+3],
top.all_delay_specific[1*`m+4],
top.all_delay_specific[1*`m+5],
top.all_delay_specific[1*`m+6],
top.all_delay_specific[1*`m+7],
top.all_delay_specific[1*`m+8],
top.all_delay_specific[1*`m+9]);
$display("%0h %0h %0h %0h %0h %0h %0h %0h %0h %0h",
top.all_delay_specific[2*`m+0],
top.all_delay_specific[2*`m+1],
top.all_delay_specific[2*`m+2],
top.all_delay_specific[2*`m+3],
top.all_delay_specific[2*`m+4],
top.all_delay_specific[2*`m+5],
top.all_delay_specific[2*`m+6],
top.all_delay_specific[2*`m+7],
top.all_delay_specific[2*`m+8],
top.all_delay_specific[2*`m+9]);
$display("%0h %0h %0h %0h %0h %0h %0h %0h %0h %0h",
top.all_delay_specific[3*`m+0],
top.all_delay_specific[3*`m+1],
top.all_delay_specific[3*`m+2],
top.all_delay_specific[3*`m+3],
top.all_delay_specific[3*`m+4],
top.all_delay_specific[3*`m+5],
top.all_delay_specific[3*`m+6],
top.all_delay_specific[3*`m+7],
top.all_delay_specific[3*`m+8],
top.all_delay_specific[3*`m+9]);
$display("%0h %0h %0h %0h %0h %0h %0h %0h %0h %0h",
top.all_delay_specific[4*`m+0],
top.all_delay_specific[4*`m+1],
top.all_delay_specific[4*`m+2],
top.all_delay_specific[4*`m+3],
top.all_delay_specific[4*`m+4],
top.all_delay_specific[4*`m+5],
top.all_delay_specific[4*`m+6],
top.all_delay_specific[4*`m+7],
top.all_delay_specific[4*`m+8],
top.all_delay_specific[4*`m+9]);
$display("%0h %0h %0h %0h %0h %0h %0h %0h %0h %0h",
top.all_delay_specific[5*`m+0],
top.all_delay_specific[5*`m+1],
top.all_delay_specific[5*`m+2],
top.all_delay_specific[5*`m+3],
top.all_delay_specific[5*`m+4],
top.all_delay_specific[5*`m+5],
top.all_delay_specific[5*`m+6],
top.all_delay_specific[5*`m+7],
top.all_delay_specific[5*`m+8],
top.all_delay_specific[5*`m+9]);
$display("%0h %0h %0h %0h %0h %0h %0h %0h %0h %0h",
top.all_delay_specific[6*`m+0],
top.all_delay_specific[6*`m+1],
top.all_delay_specific[6*`m+2],
top.all_delay_specific[6*`m+3],
top.all_delay_specific[6*`m+4],
top.all_delay_specific[6*`m+5],
top.all_delay_specific[6*`m+6],
top.all_delay_specific[6*`m+7],
```

```
top.all_delay_specific[6*`m+8],
top.all_delay_specific[6*`m+9]);
$display("%0h %0h %0h %0h %0h %0h %0h %0h %0h %0h",
top.all_delay_specific[7*`m+0],
top.all_delay_specific[7*`m+1],
top.all_delay_specific[7*`m+2],
top.all_delay_specific[7*`m+3],
top.all_delay_specific[7*`m+4],
top.all_delay_specific[7*`m+5],
top.all_delay_specific[7*`m+6],
top.all_delay_specific[7*`m+7],
top.all_delay_specific[7*`m+8],
top.all_delay_specific[7*`m+9]);


//    generate_mapping_cases;
      $display("delay_combs: count=%0d",count);

//    combinations_task(num_partitions,count);

      job_mixes(num_partitions,count);

      count=count+1;

      end // order[4]
    end // order[3]
   end // order[2]
  end // order[1]
 end // order[0]
end // task

endtask
```

B.9 job mixes task

```
task job_mixes;
input [`m_bits-1:0]num_partitions;
input [31:0] count;

integer i,j,k;
integer sum;
integer index[0:`m-1];
integer ending[0:`m-1];
integer partdex;
reg all_nonzero;

begin // task

for (i=0; i<`m; i=i+1)
      begin
      if (i<num_partitions) ending[i]=1;
      else ending[i]=0;
//    $display("ending[%0d]=%0d",i,ending[i]);
      end


for (index[0]=0; index[0]<=ending[0]*`mixsum; index[0]=index[0]+`mixinc)
 begin
 for (index[1]=0; index[1]<=ending[1]*`mixsum; index[1]=index[1]+`mixinc)
  begin
  for (index[2]=0; index[2]<=ending[2]*`mixsum; index[2]=index[2]+`mixinc)
   begin
   for (index[3]=0; index[3]<=ending[3]*`mixsum; index[3]=index[3]+`mixinc)
    begin
    for (index[4]=0; index[4]<=ending[4]*`mixsum; index[4]=index[4]+`mixinc)
     begin
     for (index[5]=0; index[5]<=ending[5]*`mixsum; index[5]=index[5]+`mixinc)
      begin
      for (index[6]=0; index[6]<=ending[6]*`mixsum; index[6]=index[6]+`mixinc)
       begin
       for (index[7]=0; index[7]<=ending[7]*`mixsum; index[7]=index[7]+`mixinc)
        begin
        for (index[8]=0; index[8]<=ending[8]*`mixsum; index[8]=index[8]+`mixinc)
         begin
         for (index[9]=0; index[9]<=ending[9]*`mixsum; index[9]=index[9]+`mixinc)
              begin
              sum=0;
              all_nonzero=`TRUE;
              for (i=0; i<num_partitions; i=i+1)
                   begin
                   sum=sum+index[i];
//                   if (index[i]==0) all_nonzero=`FALSE;
                   end
//            if (sum==`mixsum && all_nonzero==`TRUE && index[0]=10 && index[1]=20 && index[2]==70)
//            if (sum==`mixsum || (index[0]==30 && index[1]==30 && index[2]==30))
              if (sum==`mixsum)
                   begin
                     $display("");
                   $display("%0d %0d %0d",index[0],index[1],index[2]);
                   for (k=0; k<`p;k=k+1)
                     begin
                     partdex=0;
                     for (i=0; i<`m;i=i+1)
                       begin
                       if (generate_mapping_cases.part_end[partdex]<i)
                         begin
```

```
                    partdex=partdex+1;
                    end
//                  $display("module=%0d partdex=%0d and index[partdex]=%0d",i,partdex,index[partdex]);
                    top.all_delay_specific[k*`m+i]=(index[partdex]*top.temp_all_delay_specific[k*`m+i])/`mixsum;
                  end // for i
                end // for k
$display("%0h %0h %0h %0h %0h %0h %0h %0h %0h %0h",
top.all_delay_specific[0*`m+0],
top.all_delay_specific[0*`m+1],
top.all_delay_specific[0*`m+2],
top.all_delay_specific[0*`m+3],
top.all_delay_specific[0*`m+4],
top.all_delay_specific[0*`m+5],
top.all_delay_specific[0*`m+6],
top.all_delay_specific[0*`m+7],
top.all_delay_specific[0*`m+8],
top.all_delay_specific[0*`m+9]);
$display("%0h %0h %0h %0h %0h %0h %0h %0h %0h %0h",
top.all_delay_specific[1*`m+0],
top.all_delay_specific[1*`m+1],
top.all_delay_specific[1*`m+2],
top.all_delay_specific[1*`m+3],
top.all_delay_specific[1*`m+4],
top.all_delay_specific[1*`m+5],
top.all_delay_specific[1*`m+6],
top.all_delay_specific[1*`m+7],
top.all_delay_specific[1*`m+8],
top.all_delay_specific[1*`m+9]);
$display("%0h %0h %0h %0h %0h %0h %0h %0h %0h %0h",
top.all_delay_specific[2*`m+0],
top.all_delay_specific[2*`m+1],
top.all_delay_specific[2*`m+2],
top.all_delay_specific[2*`m+3],
top.all_delay_specific[2*`m+4],
top.all_delay_specific[2*`m+5],
top.all_delay_specific[2*`m+6],
top.all_delay_specific[2*`m+7],
top.all_delay_specific[2*`m+8],
top.all_delay_specific[2*`m+9]);
$display("%0h %0h %0h %0h %0h %0h %0h %0h %0h %0h",
top.all_delay_specific[3*`m+0],
top.all_delay_specific[3*`m+1],
top.all_delay_specific[3*`m+2],
top.all_delay_specific[3*`m+3],
top.all_delay_specific[3*`m+4],
top.all_delay_specific[3*`m+5],
top.all_delay_specific[3*`m+6],
top.all_delay_specific[3*`m+7],
top.all_delay_specific[3*`m+8],
top.all_delay_specific[3*`m+9]);
$display("%0h %0h %0h %0h %0h %0h %0h %0h %0h %0h",
top.all_delay_specific[4*`m+0],
top.all_delay_specific[4*`m+1],
top.all_delay_specific[4*`m+2],
top.all_delay_specific[4*`m+3],
top.all_delay_specific[4*`m+4],
top.all_delay_specific[4*`m+5],
top.all_delay_specific[4*`m+6],
top.all_delay_specific[4*`m+7],
top.all_delay_specific[4*`m+8],
top.all_delay_specific[4*`m+9]);
$display("%0h %0h %0h %0h %0h %0h %0h %0h %0h %0h",
top.all_delay_specific[5*`m+0],
top.all_delay_specific[5*`m+1],
top.all_delay_specific[5*`m+2],
```

```
top.all_delay_specific[5*`m+3],
top.all_delay_specific[5*`m+4],
top.all_delay_specific[5*`m+5],
top.all_delay_specific[5*`m+6],
top.all_delay_specific[5*`m+7],
top.all_delay_specific[5*`m+8],
top.all_delay_specific[5*`m+9]);
$display("%0h %0h %0h %0h %0h %0h %0h %0h %0h %0h",
top.all_delay_specific[6*`m+0],
top.all_delay_specific[6*`m+1],
top.all_delay_specific[6*`m+2],
top.all_delay_specific[6*`m+3],
top.all_delay_specific[6*`m+4],
top.all_delay_specific[6*`m+5],
top.all_delay_specific[6*`m+6],
top.all_delay_specific[6*`m+7],
top.all_delay_specific[6*`m+8],
top.all_delay_specific[6*`m+9]);
$display("%0h %0h %0h %0h %0h %0h %0h %0h %0h %0h",
top.all_delay_specific[7*`m+0],
top.all_delay_specific[7*`m+1],
top.all_delay_specific[7*`m+2],
top.all_delay_specific[7*`m+3],
top.all_delay_specific[7*`m+4],
top.all_delay_specific[7*`m+5],
top.all_delay_specific[7*`m+6],
top.all_delay_specific[7*`m+7],
top.all_delay_specific[7*`m+8],
top.all_delay_specific[7*`m+9]);
                combinations_task(num_partitions,count);
                end // sum==mixsum
            end // for 9
        end // for 8
       end // for 7
      end // for 6
     end // for 5
    end // for 4
   end // for 3
  end // for 2
 end // for 1
end // for 0
end // task
endtask
```

B.10 combinations_task task

```
task combinations_task;
   // having determined all possible assignements (which were checked
   // already as feasible) for each partition,
   // now generate all possible combinations of these assignments
   input [`m_bits-1:0]num_partitions;
   input [31:0]procorder;

   integer partition_for_module[`m-1:0];
   integer i,j,k,dex,mod,proc,part;
   reg [`lg2pfactorial-1:0]ending[0:`m-1];
   // allows flexible number of partitions in nested for loops
   reg [(`p*`m_bits)-1:0]value;
   // a value for a single assignment of consecutive modules to processors
   reg [(`m*`p)-1:0]m_assign; // p rows of module assignments;
                              // 1 if assigned, 0 if not
   reg [`m_bits-1:0]temp; // sample of value for a single processor
   reg [`m-1:0]temp1; // sample of value for a single processor
   integer index [0:`m-1];// indices for these nested for loops
   reg [`m_bits-1:0] proc_OH[0:`p-1];


   begin //task to generate all possible combinations of schedule permutation

// set up a module_to_partition[0..m-1] map
// i.e. for every module, the partition index is stored
   dex=0;
   for (i=0; i<`m; i=i+1)
      begin
      if (generate_mapping_cases.part_end[dex]<i)
       // if module in a new partition from the old one
         begin
         dex=dex+1;
          // label for partition incremented; partition boundary crossed
         end
      partition_for_module[i]=dex;
       // set up array with partition corresponding to the module index
//       $display("module %0d in partition %0d",i,partition_for_module[i]);
      end // for i

   for (i=1;i<=`m;i=i+1)
      begin
      if (i<=num_partitions)
         begin
         ending[i-1]=assign_partition.part_q_size[i-1]-1;
         end
      else
         begin
         ending[i-1]=0;
          // less partitions than modules (m partitions of 1 in worst case)
         end
//       $display("end of partition %0d is %0d",i-1,ending[i-1]);
      end

// nested for loops for each partition...0 to .part_q_size[partition].
   for (index[0]=0; index[0]<=ending[0]; index[0]=index[0]+1) // for partition 0 orderings
    begin
    for (index[1]=0; index[1]<=ending[1]; index[1]=index[1]+1) // for partition 1 orderings
     begin
     for (index[2]=0; index[2]<=ending[2]; index[2]=index[2]+1) // for partition 2 orderings
      begin
      for (index[3]=0; index[3]<=ending[3]; index[3]=index[3]+1) // for partition 3 orderings
```

```
begin
for (index[4]=0; index[4]<=ending[4]; index[4]=index[4]+1) // for partition 4 orderings
 begin
 for (index[5]=0; index[5]<=ending[5]; index[5]=index[5]+1) // for partition 5 orderings
  begin
  for (index[6]=0; index[6]<=ending[6]; index[6]=index[6]+1) // for partition 6 orderings
   begin
   for (index[7]=0; index[7]<=ending[7]; index[7]=index[7]+1) // for partition 7 orderings
    begin
    for (index[8]=0; index[8]<=ending[8]; index[8]=index[8]+1) // for partition 8 orderings
     begin
     for (index[9]=0; index[9]<=ending[9]; index[9]=index[9]+1) // for partition 9 orderings
      begin
      mod=0;
      m_assign=0;
      for (i=0;i<`p;i=i+1) proc_OH[i]=0;
      while (mod<`m) // for each of the modules in an ordering
          begin
          proc=0;
          part=partition_for_module[mod]; // update partition variable
          // value contains the numbers of consecutive modules to be assigned to processors
          value=assign_partition.permutation[(`pfactorial*part)+index[part]]; // value for single partition
          while (proc<`p)
          begin
          temp=value>>(proc*`m_bits);
            // consecutive modules processed on a single proc count
            if (temp>1) proc_OH[proc]=proc_OH[proc]+temp-1;
            while (temp > 0) // for consecutive modules on a single processor
            begin
                m_assign[`m*proc+mod]=1;
                temp=temp-1; // decrement consecutive modules count
                mod=mod+1; // increment module index
                end
            proc=proc+1;
            end // while proc
          end // while mod

      calculate_cost(m_assign,`FALSE,procorder,num_partitions);

      // task to calculate minimum maximum delay and total cost
      // for particular assignment (also keeps top 100 history)
      end // m-1
     end // 8
    end // 7
   end // 6
  end // 5
 end // 4
end // 3
end // 2
end // 1
end // 0

calculate_cost(m_assign,`TRUE,procorder,num_partitions);
    end //task

endtask // combinations_task
```

B.11 calculate_cost task

```
task calculate_cost;
input [(`m*`p)-1:0] assignment;
input last;
input [31:0]procorder;
input [`m_bits-1:0]num_partitions;

integer i,j,k;
reg [(`m*`p)-1:0] temp;
integer delay[0:(`p*`m)-1];
reg terminate;
integer proc, mod, temp1, temp2;
 integer dex, worstdex, newdex, q_size, q_size2;
// integer count,procdex,try;
reg [99:0]check;
time totalsum, maxprocsum, old_size;
time procsum[0:`p-1];
time size[0:`max_top100_q_size-1];
time holding[0:`max_top100_q_size-1];
reg [(`m*`p)-1:0]top100_assignments[0:`max_top100_q_size-1];
reg [31:0]proc_ordering[0:`max_top100_q_size-1];
reg [63:0]sum_total[0:`max_top100_q_size-1];
// reg [(`p_bits*`p)-1:0]temp_ordering;
// reg [`p_bits-1:0]p_order[0:`p-1];
// integer order [0:`p-1];
reg [(`m*`p)-1:0] temp_assignment;
reg [`p_bits-1:0] assigned[0:`m-1];
 reg [(`m*`lg2inc)-1:0]mix_ratio[0:`mix_assign_q_size-1];
 reg [`lg2inc-1:0]ratio_temp;
 reg [(`m*`p)-1:0]mix_assignment[0:`mix_assign_q_size-1];

begin // task
 for (order[0]=0; order[0]<`p; order[0]=order[0]+1)
  begin
  for (order[1]=0; order[1]<`p-1; order[1]=order[1]+1)
   begin
   for (order[2]=0; order[2]<`p-2; order[2]=order[2]+1)
    begin
    for (order[3]=0; order[3]<`p-3; order[3]=order[3]+1)
     begin
     for (order[4]=0; order[4]<`p-4; order[4]=order[4]+1)
       begin
       if (assignment_last!=assignment) $display("assignment=%h last=%0b",assignment,last);
       assignment_last=assignment;

       for (i=0; i<`p; i=i+1)
            begin
            Sq_add(`working2_q_id,i,temp1,temp2);
            end
//     Sdisplay("");
       for (procdex=0; procdex<`p; procdex=procdex+1)
            begin
            Sq_exam(`working2_q_id,`q_length,q_size,temp1);
            for (j=0; j<q_size; j=j+1)
                 begin
                 Sq_remove(`working2_q_id,try,temp1,temp2);
                 if (j != order[procdex])
                      begin
                      Sq_add(`working2_q_id,try,temp1,temp2);
                      end
                 else
                      begin
```

```
                    p_order[procdex]=try;
                    end
                end // for j
            end // for procdex

        terminate=`FALSE;
        totalsum=0;
        proc=0;
        while (proc<`p && terminate==`FALSE)
            begin
            mod=0;
            procsum[proc]=0;
            while(mod<`m && procsum[proc]<=top.worstdelay)
                begin
                if (assignment[proc*`m+mod]==1)
                    begin
    //                  procsum[proc]=procsum[proc]+top.all_delay_specific[(p_order[proc]*`m)+mod];
                            procsum[proc]=procsum[proc]+top.all_delay_specific[(proc*`m)+mod]+(`inter_proc_OH*(`m-num_partitions-
combinations_task.proc_OH[proc]));
                    end
            mod=mod+1;
            end
        if (procsum[proc]>top.worstdelay)
            terminate=`TRUE;
        else
            begin
            totalsum=totalsum+procsum[proc];
            proc=proc+1;
            end
        end

    if (terminate==`FALSE)
        begin // terminate FALSE
        maxprocsum=0;
//      temp_ordering=0;
        for (i=0; i<`p; i=i+1)
            begin
            if (procsum[i]>maxprocsum) maxprocsum=procsum[i];
//          temp_ordering=(temp_ordering<<`p_bits) | p_order[i];
            end
        if (maxprocsum<top.bestdelay)
            begin
            top.bestdelay=maxprocsum;
            $display("calc: found a better arrangement with max delay of %0h",top.bestdelay);
            end

        $q_exam(`top100_q_id,`q_length, q_size,temp1);
        if (q_size == `max_top100_q_size)
            begin
            $q_remove(`top100_q_id,worstdex,temp1,temp2);
            q_size=q_size-1;
            newdex=worstdex;
//          $display("calc: q full with worstdex=%0d (which has been removed)",worstdex);
            // remove worst item to be replaced by new one
            end
        else
            begin
            newdex=q_size;
//          $display("calc: q not full and adding dex=%0d",q_size);
            end
        i=0;
        while (i<q_size && terminate==`FALSE)
            begin
            $q_remove(`top100_q_id,dex,temp1,temp2);
//          $display("calc: removing dex=%0d for examination i=%0d",dex,i);
```

```
                holding[i]=dex;
                if (maxprocsum > size[dex] || (maxprocsum==size[dex] && totalsum > sum_total[dex]) )
                    begin
                    $q_add(`top100_q_id,dex,temp1,temp2);
                    $q_add(`top100_q_id,newdex,temp1,temp2);
                    size[newdex]=maxprocsum;
                    proc_ordering[newdex]=procorder;
                    sum_total[newdex]=totalsum;
                    top100_assignments[newdex]=assignment;
                    terminate=`TRUE;
                    end
                i=i+1;
                end
            if (terminate==`FALSE)
                begin // item is the best yet add it in 1st position
                $q_add(`top100_q_id,newdex,temp1,temp2);
//              $display("calc:  adding newdex=%0d and its the best yet",newdex);
                size[newdex]=maxprocsum;
                proc_ordering[newdex]=procorder;
                sum_total[newdex]=totalsum;
                top100_assignments[newdex]=assignment;
                end
            else
                begin
                i=i-1;
                // when inserting, added back an extra item already
                end
            while ( i > 0)
                begin
                i=i-1;
                // add item back to LIFO queue
                $q_add(`top100_q_id,holding[i],temp1,temp2);
//              $display("calc: i=%0d added back old dex=%0d with a size of %0h",i,holding[i],size[holding[i]]);
                end

            end // main terminate FALSE

        $q_exam(`top100_q_id, `q_length, q_size,temp1);
        if (q_size == `max_top100_q_size)
            begin
            $q_remove(`top100_q_id,worstdex,temp1,temp2);
            $q_add(`top100_q_id,worstdex,temp1,temp2);
            top.worstdelay=size[worstdex];
            // worstdelay stays at infinity until q full
            end

      end // 4
    end // 3
   end // 2
  end // 1
 end // 0

if (last==`TRUE)
      begin
//    check=0;
      for (i=0;i<q_size;i=i+1)
            begin
            $q_remove(`top100_q_id,dex,temp1,temp2);
            holding[i]=dex;
                if (i>=`max_top100_q_size-10) $display("TOP100: (%0d) dex=%0d size=%0h vector=%h procorder=%0d total=%0h",q_size-
i,dex,size[dex],top100_assignments[dex],proc_ordering[dex],sum_total[dex]);
            temp_assignment=top100_assignments[dex];
            for (j=0; j<`p; j=j+1)
                begin
                for (k=0; k<`m;k=k+1)
```

```
                    begin
                    if (temp_assignment[j*`m+k]==1)
                        assigned[k]=j;
                    end
                end
            if (i>=`max_top100_q_size-10)
                begin
                if (i==`max_top100_q_size-1)
                        begin
                        $display ("(RESRES:%0d %0d %0d: %0d %0d  %0d %0d %0d %0d  %0d %0d %0d %0d",
                job_mixes.index[0],
                     job_mixes.index[1],
                job_mixes.index[2],
               assigned[0],assigned[1],assigned[2],assigned[3],
                assigned[4],assigned[5],assigned[6],assigned[7],
                assigned[8],assigned[9]);
                        $q_exam(`mix_assign_q_id,`q_length,q_size2,temp2);
                        mix_assignment[q_size2]=top100_assignments[dex];
                        mix_ratio[q_size2]=0;
                        for (k=0;k<num_partitions;k=k+1)
                            begin
                            ratio_temp= job_mixes.index[k];
                            mix_ratio[q_size2]=(mix_ratio[q_size2]<<`lg2inc) | ratio_temp;
                            end
                        $display("RESRES                    calc:                    mix_ratio[%0d]=%0h                    and
mix_assignement[%0d]=%h",q_size2,mix_ratio[q_size2],q_size2,mix_assignment[q_size2]);
                        $q_add(`mix_assign_q_id,q_size2,temp1,temp2);
                        end
                else
                        begin
                        $display ("(RES:%0d %0d %0d: %0d %0d  %0d %0d %0d %0d  %0d %0d %0d %0d",
                job_mixes.index[0],
                job_mixes.index[1],
                job_mixes.index[2],
                assigned[0],assigned[1],assigned[2],assigned[3],
                assigned[4],assigned[5],assigned[6],assigned[7],
                assigned[8],assigned[9]);
                        end
                end
            end
        top.worstdelay=`infinity; top.bestdelay=`infinity;
        end // if last

end // task

endtask // task
```

B.12 get_size function

```
function [`m_bits:0]get_size;
   input [`m-1:0]user_vector;
   input [`m_bits-1:0]start;
   input direction;

   reg [`m_bits:0]result;
   reg [`m:0]temp;
   reg [((`m+1)*`m_bits)-1:0]ending;
   // reset to min(`m,`p) at start of each partition
   // and then partition_size/2, partition_size/3,....
   // until next partition reached
   reg [((`m+1)*`m_bits)-1:0]index;

   begin
   result=1; // count terminating 1
   temp=user_vector;
   temp=temp<<1;
   temp[`m]=1;          // guaranteed end of partition in last bit
   temp[0]=1;
   if (direction==`forwards) start=start+1; // skip over
   while (temp[start]==0) // count as long as 0's encountered from specified start
      begin
      result=result+1;
      if (direction==`forwards) temp=temp>>1;
      if (direction==`backwards) temp=temp<<1;
      end
//   $display("final result=%0d",result);
   get_size=result;
   end
endfunction
```

B.13 min function

```
function [(`m_bits+`p_bits)-1:0]min;
   input [`m_bits-1:0]partition_size;
   input [`p_bits-1:0]processor_count;

   reg [(`m_bits+`p_bits)-1:0]result;
   begin
   if (partition_size<=processor_count) result=partition_size;
   else result=processor_count;
//   $display("min: in=%0d and %0d and out=%0d",partition_size, processor_count,result);
   min=result;
   end
endfunction
```

B.14 generate.v

```
/* generate.v */

initial
      begin
      #100 -> traffic_generate1;
      j_q_id=0;
      end

always @traffic_generate2
      begin
      -> traffic_generate1;
      end

always @traffic_generate1
      begin
      next_packet = $dist_exponential(seed_traffic, `traffic_mean);
      #next_packet
//    #200
//    #400
//    #600
//    #800
//    #1600
//    #100
      // generate next packet at a random time later
            begin
            -> traffic_generate2;
            end

      if (sys_ready == `TRUE && total_jobs < `total_jobs)
            begin // system ready
            $q_remove(`job_q_id, job_id, j_var, job_status);
            if (job_status == `q_empty) // need to generate new job id
                  begin
if (`silent == `FALSE) $display("generate.v: Creating new job_id %h at time %h",jobdex, $time);
                  wakeup[jobdex]=`TRUE;
                  processing_state[jobdex]=0;
                  temp3=$time-temp2;
//                $display("generate.v: @@@@@ job %h at time %h; %h delta",jobdex, $time, temp3);
                  temp2=$time;
                  jobdex=jobdex+1;
                  if (jobdex==`j)
                        begin
                        $display ("generate.v: ERROR - more than %d job ids required",`j);
                        $finish;
                        end
                  end
            else                    // job in job_id queue
                  begin
//                $display("generate.v: Using recycled job id=%h at time %h",job_id,$time);
                  wakeup[job_id]=`TRUE;
//                check_q(`job_q_id);
                  processing_state[job_id]=0;
                  end
            end
      else  // system not ready or `total_jobs processed
            begin
            if (total_jobs >= `total_jobs)
                  begin
                  if (total_jobs==`total_jobs && `silent==`FALSE) $display("generate.v: Stopped generating jobs at %0d (%0d generated)",$time,
generated_jobs);
                  end
```

```
            else
                begin
if (`silent == `FALSE)$display("generate.v: @@@@@@@@@@@@@@@@@@@@ Offered load not accepted at time %h", $time);
                end
            end
        end

always @ (wakeup or posedge kick_me)
        begin
//      twakeup=get_bit_id(wakeup);
//      $display("generate.v: wakeup=%0h (first bit set =%0d) at time %0d", wakeup, twakeup, $time);
        get_job_id(wakeup, round_robin, j_var2);
        // function gets earliest bit set and returns its bit location
        // otherwise returns `j+1
        if (j_var2 != `j+1)
            begin
            // generate packet with job_id of j_var2
            passthrough[j_var2]=`FALSE;
//          $display("generate.v: passthrough bit %d reset at time %0d",j_var2, $time);
            if ($q_full(`seq_q_id,seq_status))
                begin
//              $display ("generate.v: @@@@@sequencing queue full");
                end
            else
                begin
                twakeup=get_bit_id(wakeup);
//              $display("generate.v: (0) wakeup=%0h (first bit set=%0d) j_var2=%0d and seq_status=%0d at time %0d", wakeup, twakeup,
j_var2, seq_status, $time);
                $q_add(`seq_q_id,j_var2,j_var,seq_status);
                wakeup[j_var2]=0;
                // initiates always block until all 0's in wakeup
                end
            end
        end

always #1 clk2 = ~clk2;

always @ (posedge clk2)
        begin
        $q_exam(`seq_q_id, 1, j_var3, gen_status);
        while (j_var3 != 0)
            begin
            #1 sig_in=`FALSE;
            $q_remove(`seq_q_id, job_id, j_var1, gen_status);
            $q_exam(`seq_q_id, 1, j_var3, gen_status);
            id_in=job_id;
            #1 sig_in = `TRUE;

            if (processing_state[job_id] == 0)
                begin
                if (generated_jobs==0) first_job_time=$time;
                if (j_q_id[id_in]==0)
                    begin
                    generated_jobs=generated_jobs+1;
                    j_q_id[id_in]=1;
                    end
if(`silent==`FALSE)$display("generate.v:((((Sending job %0d into system at time %0d", id_in, $time);
                birth_time[job_id]=$time;
                user_delay[job_id]=0;
                proc_delay[job_id]=0;
                end
            end
            if (wakeup!=0)
                begin
                kick_me=`FALSE;
```

```
                #1 kick_me=`TRUE;
                end
        end

//%% always @ (posedge sig_in) $display("generate.v:Sig_in=%b and id_in=%h at time %d",sig_in, id_in, $time);

`include "get_job_id_task.v"
`include "check_q_task.v"
```

B.15 get_job_id_task task

```
/* get_job_id_task.v */
task get_job_id;
      // finds 'next' bit which is set and then returns an index of the bit
      // using a round_robin strategy.  If no bit set, function returns
      // a value of `j+1
      input [`j:0]word;
      inout [`j_bits-1:0]round_robin;
      output [`j_bits:0]result;

      reg [`j_bits-1:0]round_robin;
      reg [`j_bits:0]result;

      begin  // task
//    $display("get_job_id_task.v: input word=%b and round_robin=%h",word, round_robin);
      result=(round_robin)%(`j+1); // make sure round_robin in range
      if (word[result] == `FALSE)
            begin
            result=(result+1)%(`j+1);
            while ((word[result] == `FALSE) && (result != round_robin%(`j+1)))
                  begin
//                $display("get_job_id_task.v:word=%b and result=%d",word, result);
//                $display ("get_job_id_task.v: (1) result=%h",result);
                  result=(result+1)%(`j+1);
//                $display ("get_job_id_task.v: (2) result=%h",result);
//                $display("word[result]=%b result=%h round_robin/(`j+1)=%h",word[result],result,round_robin%(`j+1));
                  end
            if (result == round_robin%(`j+1)) result=`j+1;
            end
      round_robin=(round_robin+1)%(`j+1);
//    $display ("get_job_id_task.v: result=%h and round_robin=%h",result, round_robin);
      end // task
endtask
```

B.16 check_q_all task

```
/* check_q_task.v */
task check_q_all;

integer i, j, q_var, q_status;
integer q_l[0:`p-1];

      begin // task check_q_all

      for (j=0; j<=`p-1; j=j+1)
            begin
            $q_exam(j, 1, q_var, q_status);
            q_l[j]=q_var;
            end
      end  // task check_q_all

endtask
```

B.17 check_q task

```
task check_q;
      input [`p_bits+3:0]q_id;

      reg [`j-1:0]temp;
      reg [(`j*`p)-1:0]all_jobs;
      integer i, q_var, q_status, num_bits_set;
      integer temp2, tempid;

      begin// check_q task
      if (q_id==`override_q_id) top.override_repeat=`FALSE;
      $q_exam(q_id, 1, q_var, q_status);
      if (q_var != 0)
            begin
            for (i=0; i<=q_var-1; i=i+1)
                  begin
                  $q_remove(q_id,tempid, temp2, q_status);
                  if (q_id ==`job_q_id)
                        begin
                        $display("check_q_task.v: Job_id_q has job id %0d at time %0d",tempid, $time);
                        end
                  else if (q_id == `seq_q_id)
                        begin
                        $display("check_q_task.v: Seq_id_q has job id %0d at time %0d",tempid, $time);
                        end
                  else if (q_id == `output_q_id)
                        begin
                        $display("check_q_task.v: Output_id_q has job id %0d at time %0d",tempid, $time);
                        end
                  else if (q_id == `override_q_id)
                        begin
                        if(top.override_id==tempid) top.override_repeat=`TRUE;
//                      $display("check_q_task.v: Override_id_q has job id %0d at time %0d",tempid, $time);
                        end
                  else // if a processor
                        begin
                        $display("check_q_task.v: Proc %0d has job id %0d at time %0d",q_id,tempid, $time);
                        end
                  $q_add(q_id, tempid, temp2, q_status);
                  end
            end
      else
            begin
//          $display ("check_q_task.v: Queue empty");
            end

      end  // check_q task
endtask// check_q
```

B.18 statistics.v

```
/* statistics.v */

initial
      begin
      seed1=1234;
      prog_module=123;
      latched_id_out=0;
      end

always @ (posedge sig_out)
      begin
//    $display("statistics.v: id_out=%0d and passthrough[%0d]=%0d at time %0d ***",id_out,id_out, passthrough[id_out], $time);
      if (passthrough[id_out] == `FALSE)
            begin
//          $display ("statistics.v: @@@@@ No processing performed for job %0d at time %0d",id_out, $time);
//          $display("statistics.v: No processing performed: id_out=%0d and passthrough[%0d]=%0d at time %0h",id_out,id_out,
passthrough[id_out],$time);
            end
            prog_module=current_module(processing_state[id_out]);
//          $display("statistics.v: id_out=%0d processing_state[%0d]=%b prog_module=%0d and
user_delay_ave[%0d]=%0d",id_out,id_out,processing_state[id_out],prog_module,prog_module,user_delay_ave[prog_module]);
//          $display("statistics.v: (x) id_out=%0d and prog_module=%0d",id_out, prog_module);
            if (prog_module < `m)// not end of processing
                  begin
//                $display("statistics.v: (0.0) id_out=%0d and prog_module=%0d",id_out, prog_module);
                  if (prog_module==0)
                        begin
                        wakeup[id_out]<=`TRUE;
//                      $q_add(`output_q_id,id_out,delay,job_status);
//                      check_q(`output_q_id);
                        end
                  else
                        begin
                        mean=user_delay_ave[prog_module];
                        seed1 = user_delay_seed[prog_module];
                        delay = $dist_exponential(seed1, mean);
//                      delay = mean;
                        user_delay_seed[prog_module]= seed1;
//                      $display("statistics.v: (0) wakeup job %0d and prog_module=%0d with delay %0d at time %0d",id_out, prog_module,
delay, $time);
                        user_delay[id_out]=user_delay[id_out]+delay;// cumulative sum recorded
//                      $display("statistics.v: user_delay[%0d]=%0d",id_out, user_delay[id_out]);
//                      twakeup=get_bit_id(wakeup);
//                      $display ("statistics.v: wakeup=%0h (first bit set=%0d",wakeup, twakeup);
                        wakeup[id_out] <= #delay `TRUE;
                        end
                  end  // if not end of processing yet
            else  // destroy token and gather statistics; processing ended
                  begin
                  if ($q_full(`job_q_id,job_status))
                        begin
                        $display ("statistics.v: Job ID queue full");
                        end
                  else
                        begin
/*
                        check_q(0);
                        check_q(1);
                        check_q(2);
                        check_q(3);
                        check_q(4);
```

```
                    check_q(5);
*/
                    total_jobs = total_jobs+1;
user_delay[id_out], total_user, proc_delay[id_out], total_proc, $time-birth_time[id_out]-user_delay[id_out]-proc_delay[id_out], total_OH);
                    total_user=total_user+user_delay[id_out];
                    total_proc=total_proc+proc_delay[id_out];
                    total_OH=total_OH+$time-birth_time[id_out]-user_delay[id_out]-proc_delay[id_out];
                    check_q_all;
                    $q_add(`job_q_id,id_out,$time,job_status);
                    j_q_id[id_out]=0;
//                  check_q(`job_q_id);
                    if (total_jobs == `total_jobs)
                        begin // end of a run
                        gather_CPU_stats;
                        output_results(`FALSE);
                        $display("statistics.v:(overrideID=%0h)  %0d  jobs  in  time  %0d  (ELAPS*PROC/2**20=%0d)",override_id,
total_jobs, $time-first_job_time, (total_proc*($time-first_job_time)>>20) );
//                      $display("statistics.v:  ####  OH*thruput=%0d  for  Override  ID=%0h",(total_proc*($time-first_job_time)>>20),
override_id);
                        $display("statistics.v:(overrideID=%0h)  OH=%0d;Proc=%0d;User=%0d",override_id,  total_OH,  total_proc,
total_user);
                        for (procdex=0; procdex<=`p-1; procdex=procdex+1)
                            begin
                            $q_exam(procdex,2,t2,t2);
                            $q_exam(procdex,3,t3,t2);
                            $q_exam(procdex,4,t4,t2);
                            $q_exam(procdex,6,t6,t2);
                            $display("statistics.v: %0d avg int_arriv=%0d; maxlen=%0d; short_wait=%0d; avg_wait=%0d",procdex,t2, t3,
t4, t6);
                            end
                        if (total_runs==`total_runs)
                          begin
                          $finish;
                          end
                        end  // end of a run

                    if (total_jobs <= (generated_jobs) && total_jobs>= `total_jobs)
                        begin  // flushing
                        j_qdex=get_bit_id(j_q_id);
                        output_results(`FALSE);
                        end   // flushing

                    if (total_jobs==generated_jobs)
                        begin  // flush complete
                        total_runs=total_runs+1;
                        total_jobs=0; generated_jobs=0;
                        j_q_id=0;
                        total_OH=0; total_user=0; total_proc=0;
                        for (i=0; i<`p; i=i+1)
                          begin
                          last_busy_time[i]=$time;
                          busy_count[i]=0;
                          end
                        for (i=0; i<`m; i=i+1)
                          begin
                          user_delay_seed[i]=`user_seed+i;
                          end
                        last_total_jobs=0;
                        epoch_count=0;
                        output_results(`FALSE);
                        override_repeat=`TRUE;
                        while (override_repeat==`TRUE)
                            begin
                            override_gen(override_id);
                            check_q(`override_q_id);
```

```
                        end
                    $q_add(`override_q_id,override_id,$time,job_status);
                    $display("");
                    $display ("statistics.v: Starting run %0d with override=%0h at time %0d",total_runs, override_id, $time);
//                  $display("statistics.v: override_id=%0h at time %0d",override_id, $time);
                    end  // flush complete
//                check_q(`job_q_id);
                  end // job queue not full
              end // destroy token and gather statistics
//          end  // else for no processing performed this time round
    end // posedge sig_out

`include "current_module_function.v"
`include "get_bit_id_function.v"
```

B.19 current_module_function function

```
/* current_module_function.v */
    function [`m_bits:0]current_module;// returns last unprocessed module
        input [`m-1:0]module_word;

        reg [`m-1:0]result;
        reg [`m-1:0]temp;

        begin
//      $display("current_module_function.v:module_word=%h",module_word);
        result=0;
        temp=module_word;
        while (temp[0])                          // count as long as 1's encountered
            begin
            result=result+1;
            temp=temp>>1;
//          $display("current_module_function.v:temp=%d and result=%d",temp, result);
            end
//      $display ("current_module: result=%h",result);
        current_module=result;
        end
    endfunction
```

B.20 get_bit_id_function function

```
/* get_bit_id_function.v */
    function [`j_bits:0]get_bit_id;// returns last unprocessed module
        input [`j-1:0]module_word;

        reg [`j_bits:0]result;
        reg [`j-1:0]temp;

        begin
        result=0;
        temp=module_word;
        while (temp[0]==0 && (result<=`j-1)) // count as long as 0's encountered and end of string not reached
            begin
            result=result+1;
            temp=temp>>1;
            end
        get_bit_id=result;
        end
    endfunction
```

B.21 override_gen_task task

```
/* override_gen_task.v */
task override_gen;
output [0:(`m_bits*`p)-1]countid;

integer moduledex, procdex;
integer seed;
integer proc;
reg [`m-1:0]toverride;
reg[`m_bits-1:0] count[0:`p-1];
reg[(`m_bits*`p)-1:0] countid;
reg at_least_one_count_zero;

      begin// override_gen_task
      at_least_one_count_zero=`TRUE;
      while (at_least_one_count_zero == `TRUE)
            begin
            at_least_one_count_zero=`FALSE;
            for (procdex=0; procdex < `p; procdex=procdex+1)
                  begin
                  count[procdex]=0;
                  end

            for (moduledex=0; moduledex < `m; moduledex=moduledex+1)
                  begin
                  seed= top.override_seed;
                  proc=$dist_uniform(seed,0,`p-1);
                  top.override_seed=seed;
                  count[proc]=count[proc]+1;
                  end

            for (procdex=0; procdex < `p; procdex=procdex+1)
                  begin
                  if (count[procdex]==0) at_least_one_count_zero=`TRUE;
                  end
            end  // while
      moduledex=0;
      countid=0;
      for (procdex=0; procdex < `p; procdex=procdex+1)
            begin
            toverride=0;
//          $display("override_gen_task.v: (1) countid=%b and count[%0d]=%0d",countid, procdex, count[procdex]);
            countid=(countid<<`m_bits) | count[procdex];
//          $display("override_gen_task.v: (2) countid=%b and count[%0d]=%0d",countid, procdex, count[procdex]);
            while(count[procdex] !=0)
                  begin
                  toverride[moduledex] =1'b1;
                  moduledex = moduledex+1;
                  count[procdex]=count[procdex]-1;
                  end
if (`silent==`FALSE)$display ("override_gen.v: RES override=%b",toverride);
            top.override[procdex]=toverride;
            end
      end  // override_gen_task

endtask
```

B.22 output_results_task task

```
/* output_results_task.v */
task output_results;
input display_on;

time last_time_called;
reg last_display_on;
integer count,i;
time CPU[0:`p-1];
time CPU_SUMSQ[0:`p-1];
time CPU_SUM[0:`p-1];
time THRU;
time THRU_SUMSQ;
time THRU_SUM;

    begin
//  $display("output_results_task.v: time=%0d and display_on=%0b and last_display_on=%0b",$time, display_on, last_display_on);
    if (last_display_on==`FALSE && display_on==`TRUE)
        begin
        count=0;
        for(i=0; i<`p; i=i+1)
            begin
            CPU_SUM[i]=0;
            CPU_SUMSQ[i]=0;
            end
        THRU_SUM=0;
        THRU_SUMSQ=0;
        $display ("    ID:time;CPU0;CPU1;CPU2;CPU3;CPU4;jobs/delta");
        end

    if(display_on==`TRUE && ($time-last_time_called)==(`epoch_width*`clk_period_div2))
        begin
        for(i=0; i<`p; i=i+1)
            begin
            CPU[i]=(100*`sigdec*top.busy_count[i]/($time-last_time_called));
            CPU_SUM[i]=CPU_SUM[i]+CPU[i];
            CPU_SUMSQ[i]=CPU_SUMSQ[i]+$rtoi($itor(CPU[i])/`sigdec*$itor(CPU[i])/`sigdec);
//  $display("CPU_SUM[%0d]=%0f and CPU_SUMSQ[%0d]=%0f",i,$itor(CPU_SUM[i])/`sigdec,i, $itor(CPU_SUMSQ[i]) );
            end
        THRU=`sigdec*(total_jobs-last_total_jobs)*`clk_period_div2*`epoch_width/($time-last_time_called);
        THRU_SUM=THRU_SUM+THRU;
        THRU_SUMSQ=THRU_SUMSQ+$rtoi($itor(THRU)/`sigdec*$itor(THRU)/`sigdec);
        $display ("DATA;%0h;%0d;%3.1f;%3.1f;%3.1f;%3.1f;%3.1f;%3.1f", top.override_id, $time,
$itor(CPU[0])/`sigdec,$itor(CPU[1])/`sigdec,$itor(CPU[2])/`sigdec,$itor(CPU[3])/`sigdec,$itor(CPU[4])/`sigdec,$itor(THRU)/`sigdec);
        count=count+1;
        end

    if (last_display_on==`TRUE && display_on==`FALSE)
        begin
        $display("MEANS;%0h;%0d;%3.1f;%3.1f;%3.1f;%3.1f;%3.1f;%3.1f;%3.1f;%3.1f;%3.1f", top.override_id, $time,
$itor(CPU_SUM[0])/count/`sigdec,$itor(CPU_SUM[1])/count/`sigdec,$itor(CPU_SUM[2])/count/`sigdec,$itor(CPU_SUM[3])/count/`sigdec,
$itor(CPU_SUM[4])/count/`sigdec, $itor(CPU_SUM[5])/count/`sigdec, $itor(CPU_SUM[6])/count/`sigdec, $itor(CPU_SUM[7])/count/`sigdec,
$itor(THRU_SUM)/count/`sigdec);
        $display("VARS;%0h;%0d;%3.1f;%3.1f;%3.1f;%3.1f;%3.1f;%3.1f;%3.1f;%3.1f;", top.override_id, $time,
($itor(CPU_SUMSQ[0])/(count-1))-($itor(CPU_SUM[0])/`sigdec*$itor(CPU_SUM[0])/`sigdec)/count/(count-1),
($itor(CPU_SUMSQ[1])/(count-1))-($itor(CPU_SUM[1])/`sigdec*$itor(CPU_SUM[1])/`sigdec)/count/(count-1),
($itor(CPU_SUMSQ[2])/(count-1))-($itor(CPU_SUM[2])/`sigdec*$itor(CPU_SUM[2])/`sigdec)/count/(count-1),
($itor(CPU_SUMSQ[3])/(count-1))-($itor(CPU_SUM[3])/`sigdec*$itor(CPU_SUM[3])/`sigdec)/count/(count-
1),($itor(CPU_SUMSQ[4])/(count-1))-($itor(CPU_SUM[4])/`sigdec*$itor(CPU_SUM[4])/`sigdec)/count/(count-1),
($itor(CPU_SUMSQ[5])/(count-1))-($itor(CPU_SUM[5])/`sigdec*$itor(CPU_SUM[5])/`sigdec)/count/(count-1),
($itor(CPU_SUMSQ[6])/(count-1))-($itor(CPU_SUM[6])/`sigdec*$itor(CPU_SUM[6])/`sigdec)/count/(count-1),
($itor(CPU_SUMSQ[7])/(count-1))-($itor(CPU_SUM[7])/`sigdec*$itor(CPU_SUM[7])/`sigdec)/count/(count-1) );
```

```
            $write("%0f", ($itor(THRU_SUMSQ)/(count-1))-($itor(THRU_SUM)/`sigdec*$itor(THRU_SUM)/`sigdec)/count/(count-1) );
            end

        last_time_called=$time;
        last_display_on=display_on;
        end

endtask
```

B.23 gather_CPU_stats task

```
/* gather_CPU_stats_task.v */
task gather_CPU_stats;

integer i;

    begin
    if(p0.s.CPUhold==`TRUE)
        begin
        top.busy_count[0]=top.busy_count[0]+$time-top.last_busy_time[0];
        top.last_busy_time[0]=$time;
        end
    if(p1.s.CPUhold==`TRUE)
        begin
        top.busy_count[1]=top.busy_count[1]+$time-top.last_busy_time[1];
        top.last_busy_time[1]=$time;
        end
    if(p2.s.CPUhold==`TRUE)
        begin
        top.busy_count[2]=top.busy_count[2]+$time-top.last_busy_time[2];
        top.last_busy_time[2]=$time;
        end
    if(p3.s.CPUhold==`TRUE)
        begin
        top.busy_count[3]=top.busy_count[3]+$time-top.last_busy_time[3];
        top.last_busy_time[3]=$time;
        end
    if(p4.s.CPUhold==`TRUE)
        begin
        top.busy_count[4]=top.busy_count[4]+$time-top.last_busy_time[4];
        top.last_busy_time[4]=$time;
        end
    end

endtask
```

B.24 proc.v

```
/* proc.v */
module proc(proc_ready_in,job_id_in, new_job_in, job_id_out, job_out,proc_ready_out);

input proc_ready_in;
input [`j_bits-1:0]job_id_in;
input new_job_in;
output [`j_bits-1:0]job_id_out;
output job_out;
output proc_ready_out;

integer qsize1, temp2;
integer qsize2, temp4;
integer qsize3, temp6;
integer rx_status, length, rx_job_id, user_var;
integer rx_q_id;
integer idle_time;
reg clk;

reg rx_q_arrival;
reg new_id_flag;
reg rx_ready;
reg proc_ready_out;

reg [`j_bits-1:0]retrieve;
reg rx_q_empty;

wire [`j_bits-1:0]job_id_out;
wire job_out;

integer delay_min     [0:`m-1];
integer delay_max     [0:`m-1];
integer delay_specific [0:`m-1];

integer delay_generic     [0:`m-1];
integer all_delay_min     [0:(`p*`m)-1];
integer all_delay_max     [0:(`p*`m)-1];
integer all_delay_specific [0:(`p*`m)-1];

parameter [`p_bits-1:0]proc_id=1;   // parameter id for this processor (over-ridden on instantiation)

scheduler #(proc_id) s (retrieve,job_id_out,job_out,rx_q_empty);

/* initialization */
initial              // initialization of variables
      begin
//    $display ("proc.v:processor %d", proc_id);
      rx_q_id=proc_id;
      $q_initialize(rx_q_id,`FIFO,`rx_q_size,rx_status);
      top.system_q_id[(proc_id*`j)+`j-1:proc_id*`j]=0;
      rx_q_empty=`TRUE;
      retrieve=0;
      clk=0;
      idle_time=0;
      rx_q_arrival=`FALSE;
      rx_ready=`FALSE;
      proc_ready_out=`TRUE;
      new_id_flag=`FALSE;
      // set up all of data structures
      delays_setup(proc_id);
      end
```

```
//always @ (top.proc0_not_ready) $display("proc.v: proc0_not_ready=%b at time %h",top.proc0_not_ready, $time);

always #2 clk=~clk;          // clock generator

always @ (posedge new_job_in)// rx msg queue is independent of processor
    begin                                    // new item added to queue
//  $display("proc.v: There00");
    if ($q_full(rx_q_id,rx_status))
        begin
        $q_exam(rx_q_id, 1, qsize3, temp6);
        $display("proc.v:@@@@@Rx queue for processor %0d is full (%0d entries) at time %0d",proc_id, qsize3, $time);
        end
    else
        begin                                // rx queue is not full
        // add timestamped entry
if (`silent==`FALSE)$display ("proc.v:@@New item (job %0d) in rx queue proc %0d time %0d",job_id_in, proc_id, $time);
        $q_exam(rx_q_id, 1, qsize1, temp2);
        if (qsize1 >= `rx_q_threshold || proc_ready_in==`FALSE)
        // reject load into system if more than a certain number of items in any queue
            begin
            proc_ready_out=`FALSE;
            if (proc_ready_out==0 && `silent==`FALSE)$display("proc.v: proc_ready_out=%0b in %0d at time %0d",proc_ready_out,
proc_id, $time);
            end
        else proc_ready_out=`TRUE;
        end
        $q_add(rx_q_id,job_id_in,$time,rx_status);
        top.system_q_id[(proc_id*`j)+job_id_in]=1;
        rx_q_empty=`FALSE;
        rx_q_arrival=`FALSE;  // was #0
        #1 rx_q_arrival=`TRUE;  // was #0
    end

always @ (posedge new_id_flag)// scheduler requests a read from rx queue
    begin
    if (rx_ready==`FALSE)
        begin
//      $display("proc.v:Scheduler %h requests a read from rx queue",proc_id);

        $q_exam(rx_q_id, 1, qsize2, temp4);
        if (qsize2 < `rx_q_threshold && proc_ready_in==`TRUE)
        // accept load into system at proc 0 once less than a certain number of items in queue
            begin
            proc_ready_out=`TRUE;
            if (proc_ready_out==0) $display("proc.v: proc_ready_out=%b in %0d at time %0d",proc_ready_out, proc_id, $time);
            end
        else proc_ready_out=`FALSE;

        $q_remove(rx_q_id, rx_job_id, user_var, rx_status);
        if (rx_status==`q_empty)
            begin
            rx_q_empty=`TRUE;
            end
        else
            begin                            // q not empty
            if (rx_status == `OK)
                begin                        // fetched item from queue successfully
                top.system_q_id[(proc_id*`j)+rx_job_id]=0;
                retrieve=rx_job_id;
//              $display ("proc.v:Item %h(%h) ready for pickup from rx queue %d at time %h",rx_job_id,retrieve,proc_id, $time);
                end
            else
                begin                        // this should never happen
                $display ("proc.v:@@@@@Error in Rx queue %d",proc_id);
                end
```

```
                end // queue not empty clause
                #0 rx_ready=`TRUE;
                wait(!new_id_flag);
                #0 rx_ready=`FALSE;
            end // rx_ready==`FALSE
        else
            begin
            $display("proc.v: Sequencing Error");
            end
        end // posedge new_id_in

`include "delays_setup_task.v"

endmodule                       // end of proc module

`include "scheduler.v"
```

B.25 delays_setup_task task

```
/* delays_setup_task.v */

task delays_setup;
    input [`p_bits-1:0]proc_id;

    integer i, j;

    integer min_delay[0:`m-1];
    integer most_min_delay[0:`m-1];
    integer max_delay[0:`m-1];
    integer most_max_delay[0:`m-1];
    integer temp_delay;

    begin // delays_setup task
    $readmemh ("fdelay_min",proc.all_delay_min);
    $readmemh ("fdelay_max",proc.all_delay_max);
    $readmemh ("fdelay_specific",proc.all_delay_specific);
    for (i=0; i<=`m-1; i=i+1)
        begin
        most_min_delay[i]=32767;
        min_delay[i]=32767;
        most_max_delay[i]=0;
        max_delay[i]=0;
        proc.delay_min[i]=proc.all_delay_min[(`m*proc_id)+i];
        proc.delay_max[i]=proc.all_delay_max[(`m*proc_id)+i];

//      proc.delay_min[i]=0; proc.delay_max[i]=0;

        proc.delay_specific[i]=proc.all_delay_specific[(`m*proc_id)+i];
// if (`silent==`FALSE)$display("delays_setup_task.v:  min[%h]=%h  and  max[%h]=%h  spec[%h]=%h  in
%h",i,proc.delay_min[i],i,proc.delay_max[i],i,proc.delay_specific[i],proc_id);
        proc.delay_generic[i]=0;
//      $display("DATAMEANSVARS Delay for (processor %0d and module %0d) =%0h",proc_id,i,proc.delay_specific[i]);
        for (j=0; j<=`p-1; j=j+1)
            begin
            proc.delay_generic[i]=proc.delay_generic[i]+proc.all_delay_specific[(`m*j)+i];
// if (`silent==`FALSE)$display("delays_setup_task.v:  delay_generic[%h]=%h  delay_specific[%h]=%h",j,  proc.delay_generic[j],(`m*j)+i,
proc.all_delay_specific[(`m*j)+i]);
            temp_delay=proc.all_delay_specific[(`m*j)+i];
            if (temp_delay >= max_delay[i])
                begin
                if (temp_delay >= most_max_delay[i])
                    begin
                    max_delay[i]=most_max_delay[i];
                    most_max_delay[i]=temp_delay;
                    end
                else
                    begin
                    max_delay[i]=temp_delay;
                    end
                end
            if (temp_delay <= min_delay[i])
                begin
                if (temp_delay <= most_min_delay[i])
                    begin
                    min_delay[i]=most_min_delay[i];
                    most_min_delay[i]=temp_delay;
                    end
                else
                    begin
                    min_delay[i]=temp_delay;
```

```
                    end
                end

            end
        proc.delay_generic[i]=proc.delay_generic[i]/p;
        proc.delay_min[i]=min_delay[i]-proc.delay_generic[i];
        proc.delay_max[i]=most_max_delay[i]-proc.delay_generic[i];

if (`silent==`FALSE)$display("delays_setup_task.v: Final delay_generic[%h]=%h",i,proc.delay_generic[i]);

//    $display ("delays_setup_task.v: proc=%0d",proc_id);
//    $display ("delays_setup_task.v: max[%0d]=%0h most_max[%0d]=%0h min[%0d]=%0h
most_min[%0d]=%0h",i,max_delay[i],i,most_max_delay[i],i,min_delay[i],i,most_min_delay[i]);
//    $display("delays_setup_task.v: Final delay_generic[%0h]=%0h",i,proc.delay_generic[i]);
//    $display("delays_setup_task.v: delay_min[%0d]=%0d(dec) delay_max[%0d]=%0h",i,proc.delay_min[i],i,proc.delay_max[i]);

        end
//    $display("delays_setup_task.v: delays (spec, min and max) have been initialized for %h", proc_id);
    end  // task
endtask
```

B.26 scheduler.v

```
/* scheduler.v */
module scheduler (retrieve,job_out,job_out_flag,rx_q_empty);

input [`j_bits-1:0]retrieve;
output [`j_bits-1:0] job_out;
output job_out_flag;
input rx_q_empty;

reg [`j_bits-1:0]job_out;
reg job_out_flag;

reg CPUhold;

reg clk;
reg more_processing_needed;
time idle, idle_count;

// event self_schedule;
// reg rself_schedule;
reg self_schedule;

integer status, rx_q_length,i,job_status;
time partcount[0:`m-1];

parameter [`p_bits-1:0]proc_id=1;

/* initialization */
initial
        begin
//      $display("scheduler.v:Scheduler %h initialized",proc_id);
        clk=0;
        idle=0;
        idle_count=0;
        CPUhold=`FALSE;
        job_out_flag=`FALSE;
        job_out=0;
        self_schedule=`FALSE; //
        for (i=0;i<`m;i=i+1) partcount[i]=0;
        $q_initialize(`jobmix_q_id+proc_id,`FIFO,`jobmix_q_size,job_status);
        end

/* wait for an interrupt from the queues */
/*
initial
        begin
        rself_schedule=0;
        end
*/

always #`clk_period_div2 clk=~clk;

// always @self_schedule rself_schedule=~rself_schedule;
//always @self_schedule $display("scheduler.v: self_schedule event in %h at time %h", proc_id, $time);

// always @ (proc.rx_q_arrival) $display("scheduler.v:rx_q_arrival in %h at time%h",proc_id,$time);
//%% always @ (self_schedule) $display("scheduler.v:self_schedule in %h at time %h",proc_id, $time);

always @ (posedge self_schedule)
        begin
        if (proc.new_id_flag==`FALSE && proc.rx_ready==`FALSE)
                begin
```

```
            proc.new_id_flag = `TRUE;
            wait (proc.rx_ready==`TRUE); // wait until message queue ready
            if (rx_q_empty == `FALSE)// new job ready
                  begin
                  # `retrieve_delay;
                  processing (retrieve, proc_id, more_processing_needed);
                  if (more_processing_needed==`TRUE)
                        begin// continue to process more modules
                        while (more_processing_needed==`TRUE)
                              begin
                              processing (retrieve, proc_id, more_processing_needed);
                              end
//                      top.passthrough[retrieve]=`TRUE;
//                      $display("scheduler.v: passthrough bit %h set at time %h",retrieve, $time);
                        end
                  job_out = retrieve;
                  job_out_flag = `FALSE;// transmit message out
                  idle_count=0;
                  if (CPUhold) $display ("scheduler.v: CPUhold on at time %h", $time);
                  wait(!CPUhold) #1 job_out_flag = `TRUE;
                  top.system_q_id[(proc_id*`j)+job_out]=0;
                  end
            else
                  begin
//                $display("scheduler.v:Self_schedule and both queues empty in %d",proc_id);
                  idle_count = idle_count + 1;
                  if (idle_count > `idle_count_threshold)
                        begin
                        idle=idle+1;
if (`silent == `FALSE)$display("scheduler.v:@@Idle epoch in processor %0d and idle=%0h (wakeup=%0h)",proc_id, idle, top.wakeup);
                        idle_count=0;
                        end
                  end
            proc.new_id_flag=`FALSE;
            wait(proc.rx_ready==`FALSE);
            self_schedule=`FALSE;
            end
      else
            begin
            $display("scheduler.v: Error simultaneous events");
            end
      end // always (main scheduler block)

always @ (posedge clk or posedge proc.rx_q_arrival)// null task counter
      begin
      self_schedule=`TRUE;
//    $display("self_schedule=%b rx_ready=%b and new_id_flag=%b",self_schedule, proc.rx_ready, proc.new_id_flag);
      end

`include "processing_task.v"

`include "CPU_task.v"

endmodule// scheduler
```

B.27 processing_task task

```
/* processing_task.v */
task processing;
        input [`j_bits-1:0]job_id;
        input [`p_bits-1:0]proc_id;
        output process_again;

        reg process_again;
        reg choose_to_process;
        reg [`m-1:0]state;
        reg job_out_flag;

        integer seed, mean;

        reg [`m_bits:0] prog_module;

        begin// processing task

        if (top.passthrough[job_id] == `FALSE)
                begin
                state = top.processing_state[job_id];// get job state
                prog_module = current_module(state);// get current module
                process_or_not(proc_id, prog_module, choose_to_process);
                if (choose_to_process == `TRUE) // not finished and processing req'd
                        begin
                        if (prog_module==0)
                                begin
//                              seed=job_id;
//                              mean=`dialtone_mean;
/*                              if($dist_exponential(top.dialtone_seed,`dialtone_mean)<=($time-top.birth_time[job_id]))
                                        begin

                                        if (top.user_retry <= `max_retry)
                                                begin
                                                $display("processing_task.v: @@@@@module 0 passthrough for job_id=%0d",job_id);
                                                top.passthrough[job_id]=`TRUE;
                                                top.user_retry=top.user_retry+1;
                                                end
                                        else
                                                begin
                                                $display("processing_task.v: Too many retries");
                                                $finish;
                                                end

                                        end
*/
                                end
                        if (top.passthrough[job_id]==`FALSE)
                                begin
                                CPU(job_id,proc_id,prog_module,state);// process waiting job
                                top.processing_state[job_id]=state;// update job state
                                process_again=`TRUE;
// if (`silent==`FALSE)$display("processing_task.v:  @@++  Processing  job  %0d  module  %0d  in  proc  %0d  at  time%0d
state=%b",job_id,prog_module, proc_id,$time, top.processing_state[job_id]);
                                if (top.user_ip_reqd[prog_module] == `TRUE)
                                        begin
                                        top.passthrough[job_id]=`TRUE;
                                        end
                                end
                        end
                else
                        begin
```

```
                    # `scheduler_passthrough
                    process_again=`FALSE;
// if (`silent==`FALSE)$display("processing_task.v: @@>> Passing through job %0d module %0d in proc %0d at time%0d",job_id,prog_module,
proc_id,$time);
                    end

                end
            else
                begin
                # `ISR_passthrough
                process_again=`FALSE;
// if (`silent==`FALSE) $display("processing_task.v: @@ >>>>__ Quick passthrough of job %0d at %0d",job_id,proc_id);

                end

            end   // processing task
endtask// processing_task

`include "current_module_function.v"

`include "process_or_not_task.v"
```

B.28 process_or_not_task task

```
/* process_or_not_task.v */
task process_or_not;
        input [`p_bits-1:0]proc_id;
        input [`m_bits:0]prog_module;
        inout choose_to_process;

        reg choose_to_process;
        reg [`m-1:0]override;
        integer delay_distance;
        integer delay_max;
        integer delay_min;
        integer q_size;
        integer temp;

        integer random, i;

        begin// process_or_not task
        if (prog_module <= `m-1)
                begin
//              $q_exam(proc_id, 1, q_size, temp);
//              if ( (q_size*8) >= `rx_q_threshold)
                if ( `TRUE == `FALSE )
                        begin
                        choose_to_process=`FALSE;
                        end
                else
                        begin
                        delay_min = proc.delay_min[prog_module];
                        delay_max = proc.delay_max[prog_module];
                        delay_distance = proc.delay_specific[prog_module] - proc.delay_generic[prog_module];
//if (`silent==`FALSE)$display("process_or_not_task.v: delay_max=%d delay_min=%d",delay_max, delay_min);
//if (`silent==`FALSE)$display("process_or_not_task.v:   (p%0d)delay_distance=%0d   delay_specific[%0d]=%0d   delay_generic[%0d]=%0d",
proc_id, delay_distance, prog_module,proc.delay_specific[prog_module], prog_module, proc.delay_generic[prog_module]);
                        if (delay_max >= delay_min)
//                      if (`TRUE==`FALSE)
                                begin
//                              if (delay_distance <= delay_min)
                                if (delay_distance < 0)
                                        begin
                                        choose_to_process=`TRUE;
if (`silent==`FALSE)$display("process_or_not_task.v: dist<min TRUE");
if ( `knox_debug == `TRUE )
$display("process_or_not1: proc %0d chooses to process module %0d",proc_id,prog_module);
                                        end
//                              else if (delay_distance >= delay_max)
                                else if (delay_distance > delay_max)
                                        begin
                                        choose_to_process=`FALSE;
if ( `knox_debug == `TRUE )
$display("process_or_not1: proc %0d chooses NOT to process module %0d",proc_id,prog_module);
if (`silent==`FALSE)$display("process_or_not_task.v: dist>max FALSE");
                                        end
                                else
                                        begin
/*
                                        random=$dist_uniform(top.uniform_seed,0,1);
//                                      $display("choose_to_process_task.v: uniform variable = %0d",random);
                                        if(random==1)
                                                begin
                                                choose_to_process=`TRUE;
if ( `knox_debug == `TRUE )
```

```
$display("process_or_not2: proc %0d chooses to process module %0d",proc_id,prog_module);
//                              $display("process_or_not_task.v: random TRUE");
                                end
                            else
                                begin
                                choose_to_process=`FALSE;
if ( `knox_debug == `TRUE )
$display("process_or_not2: proc %0d chooses NOT to process module %0d",proc_id,prog_module);
//                              $display("process_or_not_task.v: random FALSE");
                                end
*/
                            if (override[prog_module] == `TRUE)
                                begin
                                choose_to_process=`TRUE;
if ( `knox_debug == `TRUE )
$display("process_or_not3: proc %0d chooses to process module %0d",proc_id,prog_module);
//                              $display("process_or_not_task.v: override TRUE");
                                end
                            else
                                begin
                                choose_to_process=`FALSE;
if ( `knox_debug == `TRUE )
$display("process_or_not3: proc %0d chooses NOT to process module %0d",proc_id,prog_module);
//                              $display("process_or_not_task.v: override FALSE");
                                end
                        end
                    end
                else  //
                    begin
                    override = top.override[proc_id];
                    if (override[prog_module] == `TRUE)
                        begin
                        choose_to_process=`TRUE;
if ( `knox_debug == `TRUE )
$display("process_or_not4: proc %0d chooses to process module %0d",proc_id,prog_module);
//                              $display("process_or_not_task.v: override TRUE");
                                end
                    else
                        begin
                        choose_to_process=`FALSE;
if ( `knox_debug == `TRUE )
$display("process_or_not4: proc %0d chooses NOT to process module %0d",proc_id,prog_module);
//                              $display("process_or_not_task.v: override FALSE");
                                end
                    end

                proc.delay_min[prog_module]=delay_min;
                proc.delay_max[prog_module]=delay_max;

                end  // not too busy to process

            end  // if (prog_module <= `m-1)

        else choose_to_process=`FALSE; // last module already processed

        end   // process_or_not task

endtask // process_or_not task
```

B.29 CPU_task task

```
/* CPU_task.v */
task CPU;
        input [`j_bits-1:0]job_id;
        input [`p_bits-1:0]proc_id;
        input [`m_bits:0]prog_module;
        inout [`m-1:0]processing_state;

        reg [(`m-1):0]processing_state;
        reg [((`delay_bits)*`m)-1:0]delay_all_modules;
        integer delay_specific;
        integer delay_temp;

        begin// CPU task
        delay_temp=proc.delay_specific[prog_module];
        delay_specific=$dist_normal(top.CPU_seed,delay_temp,2);
//      delay_specific=delay_temp;
//      $display("CPU_task.v: delay=%0d", delay_specific);
        scheduler.CPUhold=`TRUE;
        #delay_specific;//
        top.proc_delay[job_id]=top.proc_delay[job_id]+delay_specific;
        scheduler.CPUhold=`FALSE;
        processing_state=(processing_state<<1)|1'b1;
        end   // CPU task

endtask // CPU
```

B.30 delay file

```
/* fdelay_specific*/
// Module delay (delay_bits bits per module)
// 00 01 02 03 04 05 06 07 08 09
  64 1E 1E 3C 14 28 14 32 1E 28 /* proc 0 */
  5A 32 28 32 32 1E 15 28 14 46 /* proc 1 */
  64 46 1E 1E 1E 28 1E 32 28 32 /* proc 2 */
  14 5A 28 3C 14 32 32 32 28 14 /* proc 3 */
  1E 1E 32 32 1E 14 14 32 28 5A /* proc 4 */
//  50 46 1E 1E 1E 28 1E 28 28 32 /* proc 5 */
//  34 1A 28 3C 12 32 32 32 1F 20 /* proc 6 */
// 1E 1E 32 32 1E 1F 14 32 28 40 /* proc 7 */
```

B.31 user delay file

```
/* fuser_delay_ave */
31 /* module 0 */
31 /* module 1 */
31 /* module 2 */
31 /* module 3 */
31 /* module 4 */
31 /* module 5 */
31 /* module 6 */
31 /* module 7 */
31 /* module 8 */
31 /* module 9 */
```

Appendix C

Data from Simulation shown in Fig. 6.1

Assignment Vector	Proc 0 CPU Util.	Proc 1 CPU Util.	Proc 2 CPU Util.	Proc 3 CPU Util.	Proc 4 CPU Util.	Mean	Standard Deviation
51112	49.3	6.4	5.3	8.8	22.8	1750	1975
11116	20.6	10.3	6.2	12.4	45.1	1750	768
41113	48.6	11.4	9.1	9	32.2	1790	2001
42112	47.3	17.8	5.4	9	23.3	1795	1974
52111	50.5	10.1	9.1	7.3	16.4	1825	2337
41212	47.6	10.9	14.3	9.3	24.3	1865	2043
41131	47.1	11.1	8.9	26.2	16.8	1870	2000
12115	23.2	20.8	6.9	4.6	42.8	1910	741
51121	48.5	6.3	6.3	17.5	17.3	1930	2147
11125	20.6	10.3	6.3	16.7	42.9	1930	672
61111	56.6	4.4	9.9	8	17.4	1935	2296
14212	22.8	43.3	14.9	9.7	25.3	1940	1905
41122	46.6	11	8.8	19.7	25.4	1955	2119
32113	39.2	25.6	10.3	10.1	35.3	1960	1779
12214	21	19.5	13.5	11.3	39.2	1960	675
42121	46.4	17.6	6.3	17.8	17.7	1970	2288
11134	23	11.5	7.2	31.2	39.4	1970	947
21115	30	9.3	7	4.7	43.8	1975	806
51211	48.8	6.2	16.2	8.1	17.8	1980	2202
16111	21.1	52.7	9.9	8	17.8	1980	1982
11215	21.7	10.9	12.7	4.2	43.5	1980	725
31123	39.9	13.1	7.8	23	35.8	1990	1988
14113	24	44.9	9.4	9.9	35.8	1990	1908
13114	21.3	30.7	6.7	11.1	39.8	1990	835
33211	39.2	32.9	16	8	17.9	1995	1965
14131	24.2	46	9.7	28.1	18	2000	1700
41221	46.7	11.1	15.5	18.3	18.2	2020	2276
11224	22.4	11.2	13.4	15.6	40.4	2020	657

Table C.1 Fig. 6.1 Simulation Data

Assignment Vector	Proc 0 CPU Util.	Proc 1 CPU Util.	Proc 2 CPU Util.	Proc 3 CPU Util.	Proc 4 CPU Util.	Mean	Standard Deviation
15112	22.8	50.7	6.1	10.2	26.5	2035	1958
22114	28.4	20.4	6.8	11.3	40.7	2035	867
42211	46.8	17.4	16.7	8.3	18.4	2045	2321
32212	41.2	25.9	16.5	10.2	26.6	2045	1664
11314	21.5	10.8	20.2	11.2	40.9	2045	800
12124	22.1	20.2	6.8	16	41	2055	731
11611	22	11	46.3	8.2	18.6	2060	1530
11161	21.9	10.9	6.7	58	18.7	2070	1394
14122	22.6	44.3	9.4	20.8	26.9	2075	1673
33121	39.9	33	6.3	18.7	18.7	2080	1705
11152	22.8	11.4	7.3	50	27.1	2085	1754
41311	47.3	10.9	25.9	8.6	18.9	2095	2344
31213	40.3	13.2	18.5	10.5	37.7	2095	1840
43111	46.9	21.8	10.9	8.7	18.9	2105	2376
34111	40	37.8	10.5	8.4	19	2105	1790
15121	24.9	51.9	6.5	19.1	19.1	2120	2089
15211	22.9	51.6	17.1	8.5	19.1	2125	2042
11251	27.8	13.9	15.8	46.9	19.2	2130	1236
21214	29.5	9.5	14.2	11.8	42.6	2130	707
31114	36.4	11	6.6	11	42.9	2145	952
33112	39.1	31	7.3	11.6	28	2150	1786
31132	38.9	12.8	7.7	34.4	28	2155	1978
14311	23.1	43.8	26.5	8.6	19.4	2155	1695
11143	26.7	13.3	8	45.5	38.8	2155	1589
32311	41.3	26.3	27.8	8.7	19.5	2165	1673
21124	31.5	9.7	7.2	16.9	43.7	2190	795
23113	33.8	37.2	10.6	11.3	39.6	2200	1526
24121	37.7	45.5	6.7	19.9	19.9	2205	1636
31222	39	12.8	18	22.4	28.7	2210	1950
12511	27.4	24.1	42.7	8.9	20	2230	1819
21151	33.9	10	7.5	48.1	20.1	2230	1603
24112	32	44.4	6.9	11.4	29.1	2240	1632
22213	36.8	23.6	18.4	11.4	40.3	2240	1437
14221	23.7	43.7	16.2	20.2	20.2	2245	2098
21511	32	10.5	41.7	9.1	20.2	2250	1669
11521	26.6	13.3	41.4	20.3	20.2	2250	1482
22222	36.8	23.9	18.6	22.7	29.4	2255	1305
11413	27.1	13.6	36.4	11.6	40.8	2265	1363

Table C.1 Fig. 6.1 Simulation Data

Assignment Vector	Proc 0 CPU Util.	Proc 1 CPU Util.	Proc 2 CPU Util.	Proc 3 CPU Util.	Proc 4 CPU Util.	Mean	Standard Deviation
31312	42.7	13	26.3	12.2	29.5	2270	1974
21142	34.1	10.9	8.2	40.7	29.5	2270	1588
31321	41.4	13.6	26.5	20.5	20.5	2275	1745
32221	41	25.9	17.6	20.5	20.5	2280	1659
25111	31.7	48.7	11.4	9.2	20.6	2290	1874
13231	32.5	40.7	20	32.3	20.8	2305	1736
13132	27	39.5	8.6	37.5	30	2305	1678
22141	37.4	23.7	7.9	45.3	20.8	2305	1546
31141	43.9	13.2	7.9	45.6	20.8	2310	1498
21322	37	11.5	28	23.2	30.1	2310	1328
23221	34.9	37	17.6	21	21	2330	1723
21241	36.1	10.8	16.2	44.9	21	2330	1621
11512	29.2	14.6	41.6	11.7	30.4	2330	1602
11242	27.4	13.7	16.3	40.5	30.3	2330	1577
22312	36.9	23.9	25.6	11.9	30.4	2335	1398
23212	34	35.6	17.9	12.2	30.4	2340	1655
32122	41.4	24.6	9.8	23.9	30.5	2345	1629
13213	31.1	42	21.1	11.8	42.3	2350	1576
13123	29.2	40.8	8.7	26.5	42.3	2350	1483
24211	38	42.4	18.9	9.4	21.2	2355	1846
11431	27.9	13.9	35.6	33	21.2	2355	1746
32131	40	25.5	10.2	33	21.2	2355	1624
23131	35.8	36.9	10.5	33.1	21.2	2355	1564
21223	38.1	12.3	18.1	26.9	42.4	2355	1430
21313	38.1	11.2	28.1	12.1	42.5	2360	1232
22231	38.1	24.9	19.4	33.1	21.3	2365	1379
11341	29.9	15	24.9	46.6	21.4	2375	1538
21331	34.7	11.1	27.7	33.4	21.5	2380	1459
12151	25.2	22.8	7.7	50.9	21.4	2380	1341
13312	31.3	40.7	27.6	11.9	31	2385	1793
23122	36.3	35.7	10.2	24.3	31	2385	1616
21421	36.8	11.1	35.1	21.6	21.6	2400	1749
22411	38.1	24.8	38.5	9.6	21.6	2400	1667
22123	37.3	23.3	7.8	25.4	43.2	2400	1397
22132	37.5	23.3	7.8	37	31.3	2410	1519
11323	29.2	14.6	25.3	27.1	43.4	2410	1448
12223	29.8	25.8	16.3	26.7	43.5	2415	1425
31231	41.5	13.1	18.4	33.9	21.8	2420	1813

Table C.1 Fig. 6.1 Simulation Data

Assignment Vector	Proc 0 CPU Util.	Proc 1 CPU Util.	Proc 2 CPU Util.	Proc 3 CPU Util.	Proc 4 CPU Util.	Mean	Standard Deviation
23311	34.8	35.8	29.3	9.7	21.9	2425	1794
12142	28.5	25.7	8.6	42.7	31.5	2430	1538
13222	28.1	39.1	20	24.8	31.7	2440	1668
21133	38.2	11.4	8.6	32.3	44.2	2450	1456
12412	28.7	25.8	36	12.6	31.9	2455	1595
21232	34.6	10.6	15.9	38.3	32	2460	1535
12241	26.1	23.8	16.2	47.9	22.2	2470	1396
11233	29.5	14.8	17.9	33.6	44.6	2475	1377
13321	32.8	40.6	27.3	22.6	22.5	2505	2142
11332	29.1	14.5	25.4	39.3	32.6	2505	1440
12313	27.5	25.2	28.7	14.2	45.2	2510	1239
13411	33.2	42.2	40.9	10.1	22.7	2520	2031
21412	36.3	10.3	33.6	13	32.8	2520	1365
31411	42.6	12.8	38.1	10.1	22.7	2525	2076
12322	29.7	26.6	29.3	25.5	33	2535	1383
12331	30.2	26.8	29	35.7	22.9	2540	1353
13141	26.9	37.3	8	48.9	22.9	2545	1744
12133	28.1	26	9	35.4	45.8	2545	1216
22321	38.8	24.1	26.4	23.1	23.1	2575	1808
11422	28.7	14.3	35.5	26.3	33.9	2605	1594
12421	30.7	26.7	36	24.1	24	2660	1754
12232	29.7	26.4	17.1	40.2	34.6	2660	1455

Table C.1 Fig. 6.1 Simulation Data