INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.

University Microfilms International
300 N. Zeeb Road
Ann Arbor, MI 48106

Wang, I-Yang

SIMULATION OF A MODULAR HIERARCHICAL ADAPTIVE COMPUTER ARCHITECTURE WITH COMMUNICATION DELAY

The University of Arizona M.S. 1986

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark __√__.

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy _____
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages √
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print _____
11. Page(s) _____ lacking when material received, and not available from school or author.
12. Page(s) _____ seem to be missing in numbering only as text follows.
13. Two pages numbered _____. Text follows.
14. Curling and wrinkled pages _____
15. Dissertation contains pages with print at a slant, filmed as received _____
16. Other _____

_____

_____

University
Microfilms
International

SIMULATION OF

A

MODULAR HIERARCHICAL ADAPTIVE COMPUTER ARCHITECTURE

WITH

COMMUNICATION DELAY

by

I-Yang Wang

A Thesis Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of Requirements
For the Degree of

MASTER OF SCIENCE
WITH A MAJOR IN ELECTRICAL ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 8 6

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under the rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interest of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Wang L Gang

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

BERNARD P. ZEIGLER
Professor of Electrical and
Computer Engineering

11/26/86
Date

ACKNOWLEDGMENT

I would like to express thankfulness to Professor Bernard P. Zeigler, my major advisor, for his patience and guidance in the research period.

I would also like to thank Dr. J. W. Ronzenblit and Dr. B. Soucek for their suggestions and helps. I am also grateful to the members of the Architecture Groups, Taggon Kim, Yih-Shyan Liaw, and Ming-Jehn Ferng, who consistently offered me valuable advice and comments.

Finally, I would like to dedicate this thesis to my wife, Jennie, and my parents for their enthusiastic support and encouragement.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

ABSTRACT

In this thesis, we extended an existing adaptive computer system model by introducing a new "channel" module which handles the communication delay of messages transmitted between processors. We employed a structure state concept to facilitate modular design of the model. We studied the system's performance as a function of communication delay.

We studied and compared the behaviors of different system configurations and adaptive control signal generation policies in different environments. Over 800 simulation replications were run.

In our experiments, the "pipeline" coordinator was proven to be the worst in almost all cases. A combination, using of the "multiserver" and the "divide'n'conquer" coordinators, gave best system performance at the fast packet arrival rate and the "divide'n'conquer" coordinator was best at slow packet arrival rate. In comparison of the adaptive control signal generation policies, the "Fc-root" policy was best and the "root-Fp" policy was worst. We analyzed these results and offered explanations for the observed behaviors.

CHAPTER 1

INTRODUCTION

1.1 Background

System adaptability is one of the important issues of time-critical computing systems which may require real-time response and very high throughput. An adaptive computing system is able to reconfigure its architecture by dynamically allocating or deallocating available resources to accomodate to the time changing environment. Zeigler and Reynold (1984) proposed that an optimum structural system matches its environment and will evolve to that optimum structure over time [7]. Linos (1985) introduced a modular hirerarchically coordinated adaptive architecture which applies its basic configuration recursively and results in a tree of processors [4]. Linos also designed and implemented the simulation model of this architecture. Linos' main concern is the performance of this architecture -- the maximum number of processors used over time, the average turn around time of the problems, and the through-put (number of problems solved divided by processing time).

In Linos' model, he assumed that the communication delay of transmitting problems and signals between processors are negligible. However, this is not true in the

real world; any transmission of message must suffer a delay time which varies from one system to another. Therefore, simulation results with consideration of communication delay may better reflect the accurate behaviors of the adaptive computer system in the real world.

In this work,[1] Linos' model is extended by incorporating communication delay which occurs whenever messages are transferred from one processor to another. The message transmission is modelled by an infinite capacity "channel" and any message transmission must go through a "channel" module; the message is submitted to a "channel", delayed for a constant interval, and, then, released from the "channel" to its destination processor.

1.2 Objectives

Our objectives are first to design and implement the incorporation of communication delay in Linos' model and second to study the effect of communication delay on the overall system performance, which are the number of processors used in the system, the throughput, the average packet turn around time, and the packet batch average turn around time.

---

1. This work was supported by NSF Grants DCR 8407230 and 8514348.

1.3 Assumptions

Our simulation model is designed and implemented under several assumptions. First, the "channel" module is able to transfer an unlimited number of problems or signals at the same time. Second, the communication delay is considered as a system parameter for messages transferred between processors; all messages passing through the "channel" suffer a constant delay time. However, the delay of the message transmission within a processor is not considered. Third, the fan out of each node of the tree is not greater than three. Fourth, the system does not influence its environment. Finally, the number of processors is finite but unbounded. The last three are originally found in Linos' model.

1.4 Contributions

In our modular design, modules communicate via message exchanges which must go through input and output ports. To achieve this goal, we introduced "structure state" concept in which each processor is able to identify its position in the system tree by keeping track of its children's types. This allows it to determine whether to accept or to transmit adaptive control signals.

In our experiments, over 800 simulation replications were run. We studied the system's behaviors by running each experiment five times and averaging these results to

illustrate how communication delay affects system performance. In the cases of zero communication delay, the findings of the best and the worst system configurations are consistent with Liaw's [3]. However, when the communication delay increases, the behaviors of the system change. Comparisons of the adaptive control signal generation policies' performance were also made to demonstrate the best and the worst policies. We analyzed these results and offered explanations for the observed behaviors.

CHAPTER 2

MODEL FUNCTIONAL DESCRIPTIONS

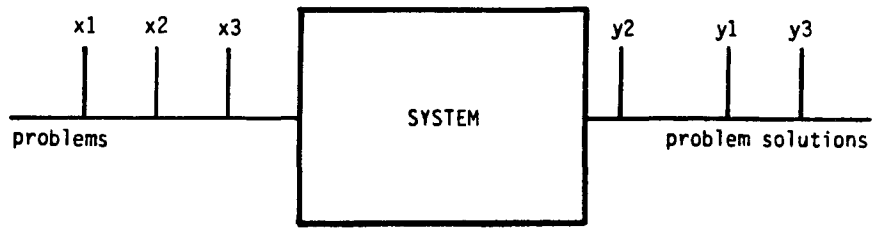
Our simulation model is extended from Linos' and some modifications are made in order to maintain its full modularity and incorporate the communication delay by adding a "channel" module. The simulation manipulations as well as the data structures which represents the system and the environment will be discussed in detail in this chapter.

2.1 General Model Description

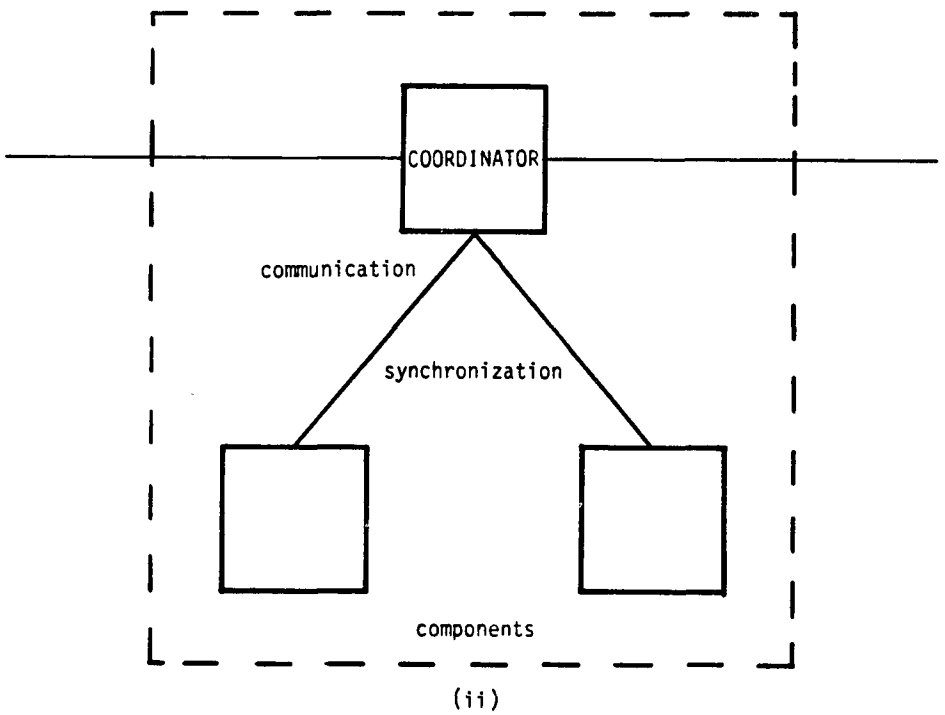
The basic sysetm can be considered as a "black box" (Figure 2.1 (i)). Problems arrive with a fixed interarrival time t and are solved by the system. The system environment is parameterized by the interarrival time t and the problem complexity. The system is a "black box" which consists of other "black boxes" and is described by the coupling of these "black boxes" and the communication delay between them.

2.1.1 Coordinator

The coordinator is responsible for directing the information flow specified by the coupling between the environment and the system components and synchronizing the activities of the components [4]. The lowest level



(i)



(ii)

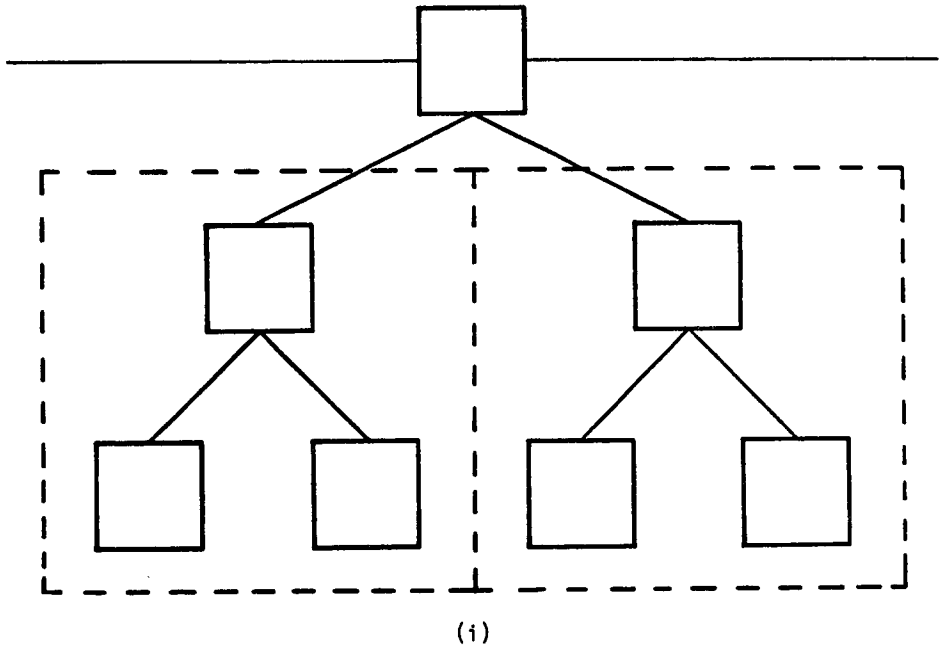
Figure 2.1 (i) A basic system. (ii) a coordinated coupled system

components are considered to be functional in nature and the coordinator is supervisory (Figure 2.1 (ii)). The structure can be applied to itself recursively (Figure 2.2 (i)) and results in a hierarchical tree (Figure 2.2 (ii)) in which the interior nodes represent coordinators and the lowest terminal leaves are functional elements.

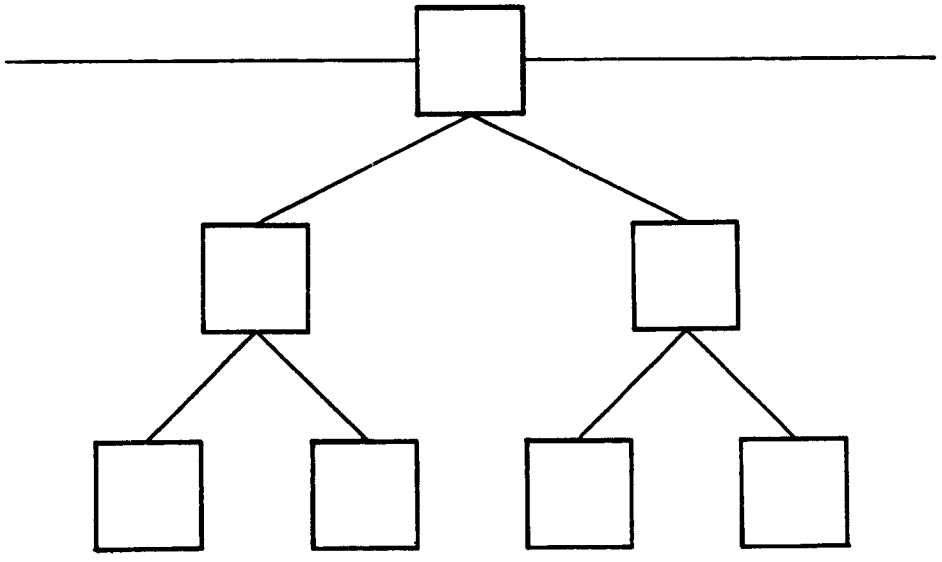
2.1.1.1 Type of Coordinators. There are three types of coordinators -- "multiserver", "pipeline", and "divide'n'conquer" [7]. These coordinators have different ways of synchronizing and communicating with their components.

"Multiserver" type coordinators (Figure 2.3 (i)) redirect the incoming problems to its lower level with less work load. When the problem is solved, it is returned to the higher level coordinator.

"Pipeline" type coordinators (Figure 2.3 (ii)) divide the incoming problems into the "lowerhalf" and the "upperhalf" problems. The "lowerhalf" is sent to its left processor. The "upperhalf" problem is sent to its right processor only under the condition that the "lowerhalf" problem is returned from the left processor and not solved. If the problem is solved, which is described by a "found" attribute of the problem, the "upperhalf" will not be sent to its right process and the problem is routed to its parent coordinator.



(i)



(ii)

Figure 2.2 (i) Recursive application of structure to components. (ii) Interpretation of result of refinement as a hierarchically coordinated system

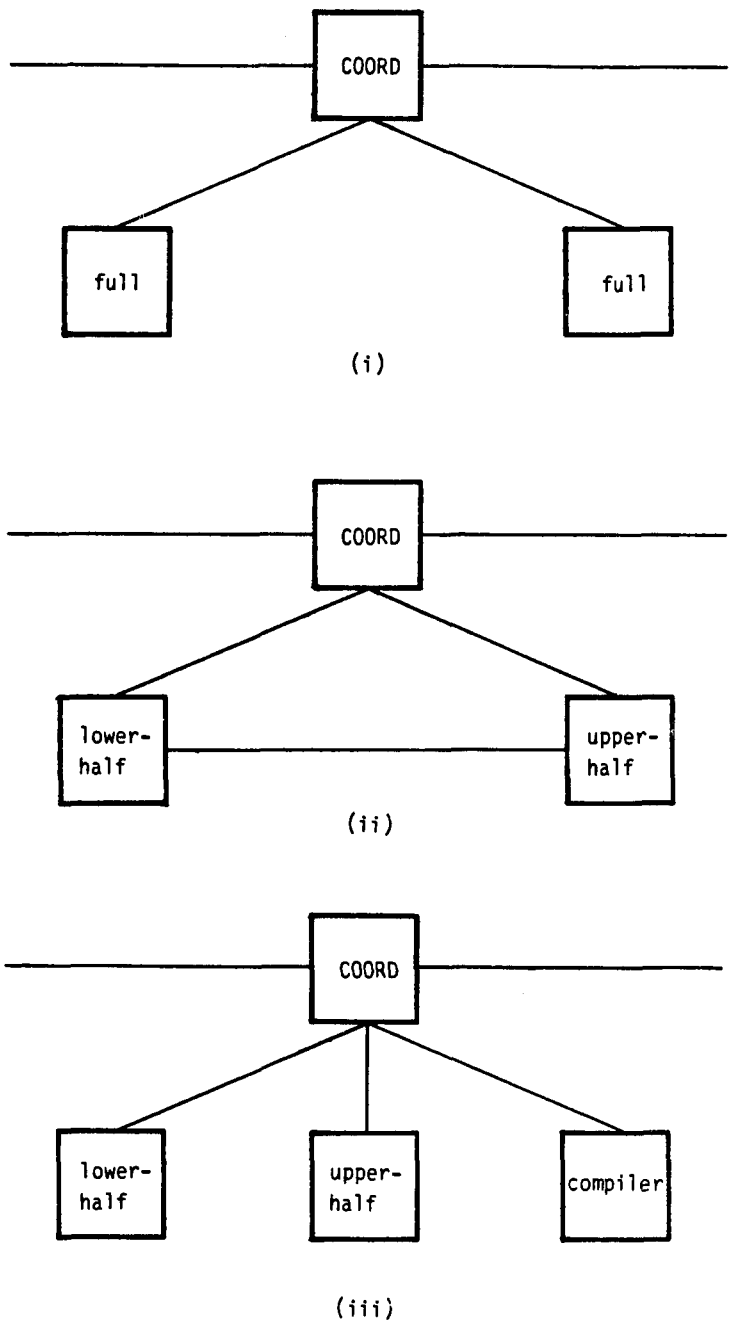


Figure 2.3 (i) "Multiserver" coordinator. (ii) "Pipeline" coordinator. (iii) "Divide'n'conquer" coordinator

"Divide'n'conquer" type coordinators (Figure 2.3 (iii)) also divide the incoming problems into the "lowerhalf" and the "upperhalf" problems. Both the "lowerhalf" and the "upperhalf" problems are sent to its left and right processors at the same time respectively. After both "half" problems are returned, they are sent to the third processor "compiler" which is responsible to compile the solutions obtained from the left and the right processors. After being compiled, the problem solution is returned to the coordinator.

2.2 Data Structure Description

In our simulation model, there are two types of tree structures containing "flexible processors" and "packets" (Figure 2.4).

2.2.1 Flexible Processor Tree

The first tree structure consists of nodes called "flexible processors" (we will use "fp" for short) and second one called "packet" which contains the problems to be solved by the system.

Each fp can be represented by a number of fields to be able to identify its state in the tree structure (Figure 2.5). An fp can be an "f-computer", that is a leaf or terminal node of the data structure, which indicates that it is an undecomposed functional element. Or, it can be a "coordinator" which is an interior fp of the tree with

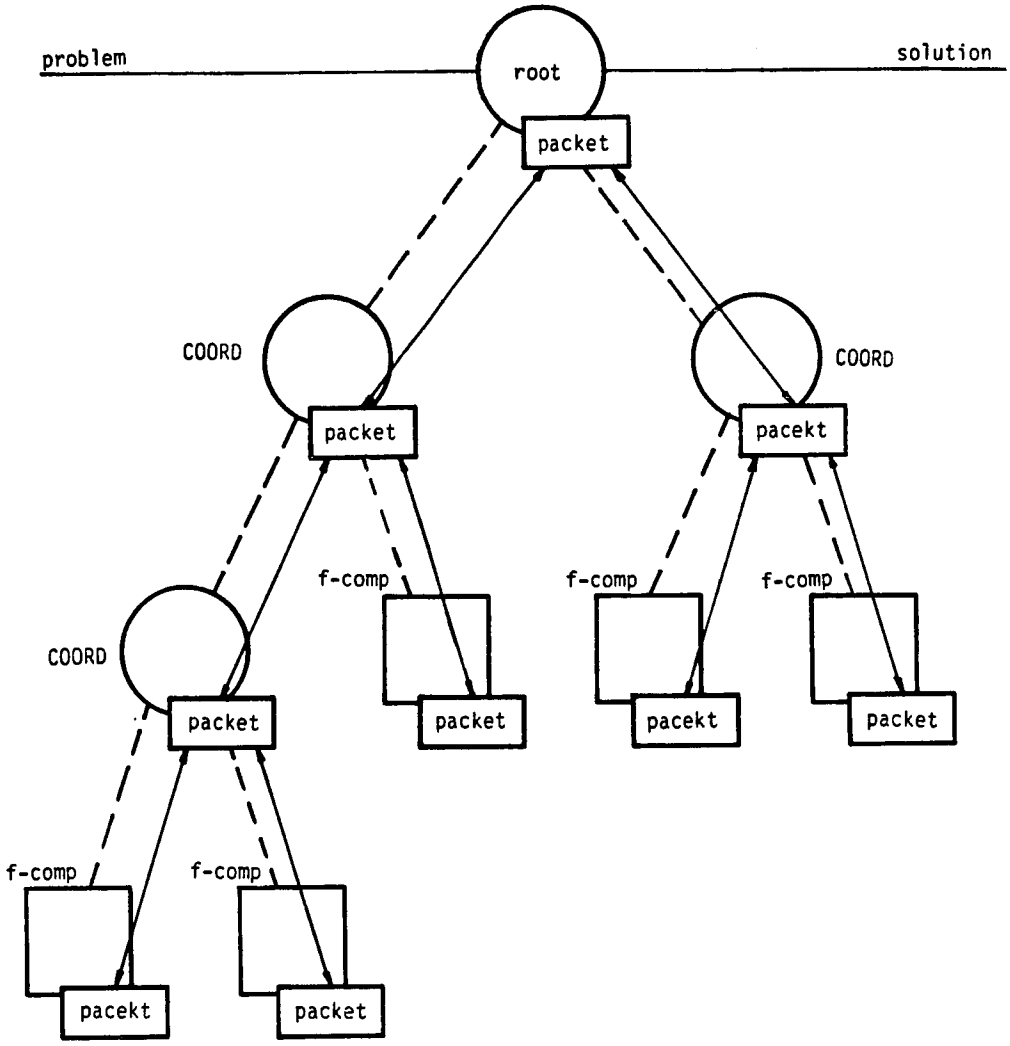


Figure 2.4 Tree structure for the flexible processors and the packets

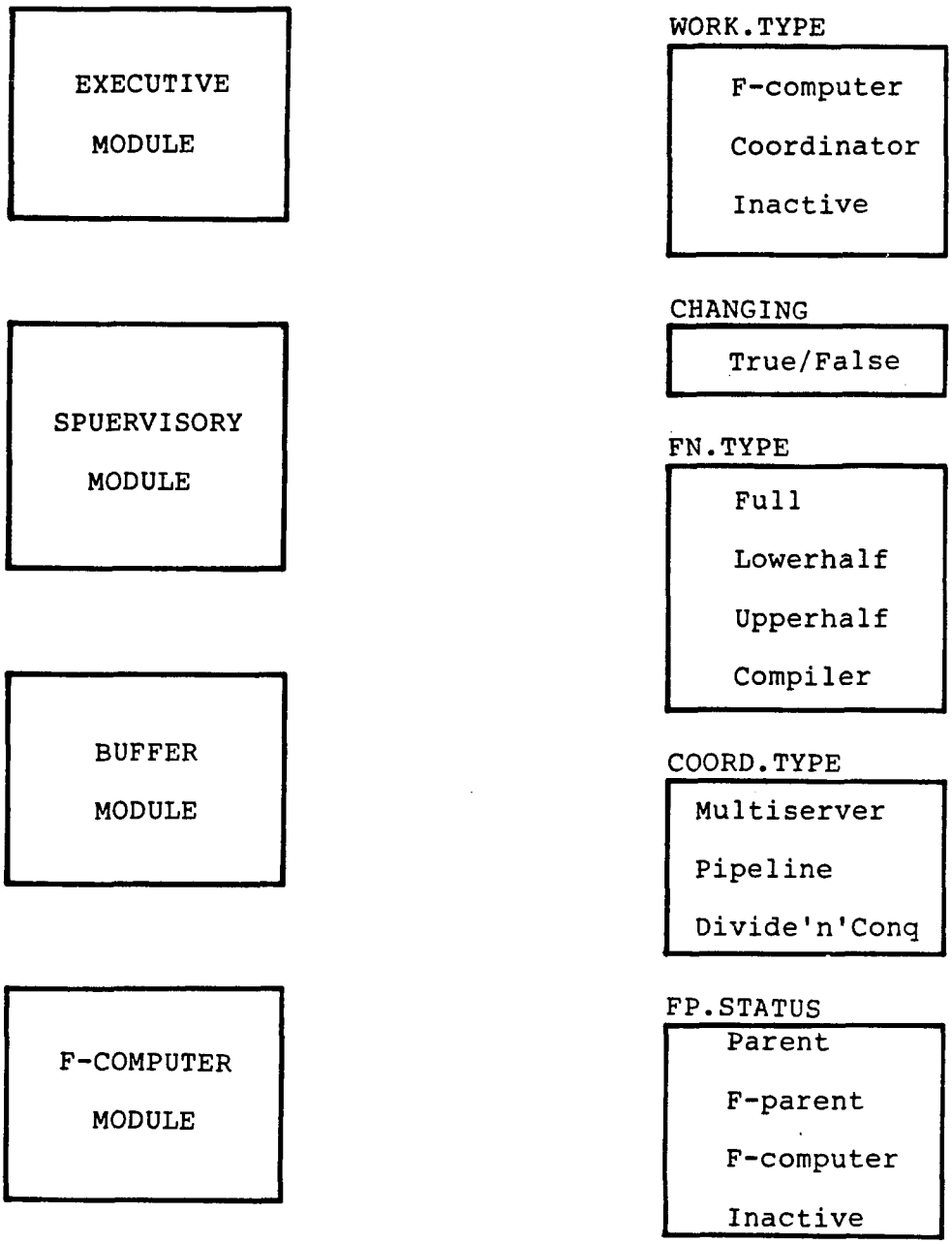


Figure 2.5 Data structure of flexible processor

supervisory responsibilities. An fp can also be "inactive" which means the fp is in an fp "pool" and is not yet allocated for use, or it is ready to be removed from the system tree. This piece information is given in the "work-type" field with range <"f-computer", "coordinator", "inactive">. To indicate the transition state of "work-type", an fp can be in "changing" mode which is indicated by the "changing" field with range <true, false>. Another field, "fn-type" (function type) with range <"lowerhalf", "upperhalf", "full", "compiler">, stands for the part of the problem that the current fp is working on. The field "coordinator-type" specifies the coordinator type of an fp; an fp can be a "multiserver", a "pipeline", or a "divide.and.conquer" type coordinator. The final field "fp-status" with range <"parent", "f-parent", "f-computer", "inactive"> indicates the relationship of this fp with the others. The meanings of "inactive" and "f-computer" are the same as in "work-type" field. An fp is an "f-parent" if both of its children are "f-computer". If an fp has at least one "f-parent" child, it is a "parent".

2.2.2 Packet Tree

The second tree data structure consists of "packets" coming into the system (Figure 2.4). Each packet has a "parent.packet", a "left.packet" and a "right.packet", which are pointers with responsibility of keeping track of the

divided packets being sent to the next lower level fp according to different coordinator types. When a coordinator receives a packet, it divides the packet into the "lowerhalf" and the "upperhalf", which, actually, are copies of the original packet. The "parent" packet stays in the coordinator's "packet.waiting" queue and the half-packets are sent to the lower level processors according to the coordinator types. The half-packets are pointed by their parent packet's "left.packet" and "right.packet" respectively; each one's "parent.packet" field points to the parent packet. The packets are also characterized by three numbers (l, u, j) which represent the problem and its complexity. The computation time T required by the problem is determined as follows:

. if j <= l, T = 0

. if l < j < u, T = j - l

. if j >= u, T = u - l

2.3 Module Descriptions

There are several modules, the "executive", the "supervisory", the "buffer", and the "f-computer", related to each fp. There is one more module "channel", which represents a transmission medium between processors. Later on, we will use capitalized words, EXECUTIVE, BUFFER, SUPERVISOR, FCOM, and CHANNEL respectively to indicate that they are modules in the fp and the system. Each module has

different functional responsibilities and duties (Figure 2.6). Inputs and outputs (I/O) of the modules in an fp will be defined at the beginning of each description, which are also classified as "external" and "internal". The "external" means that these signals are connected outside the node, i.e. some other nodes; any "external" I/O are through CHANNELs. Similarly, the "internal" I/O indicates the signal connections within an fp; they do not go through CHANNELs.

2.3.1 F-computer Module

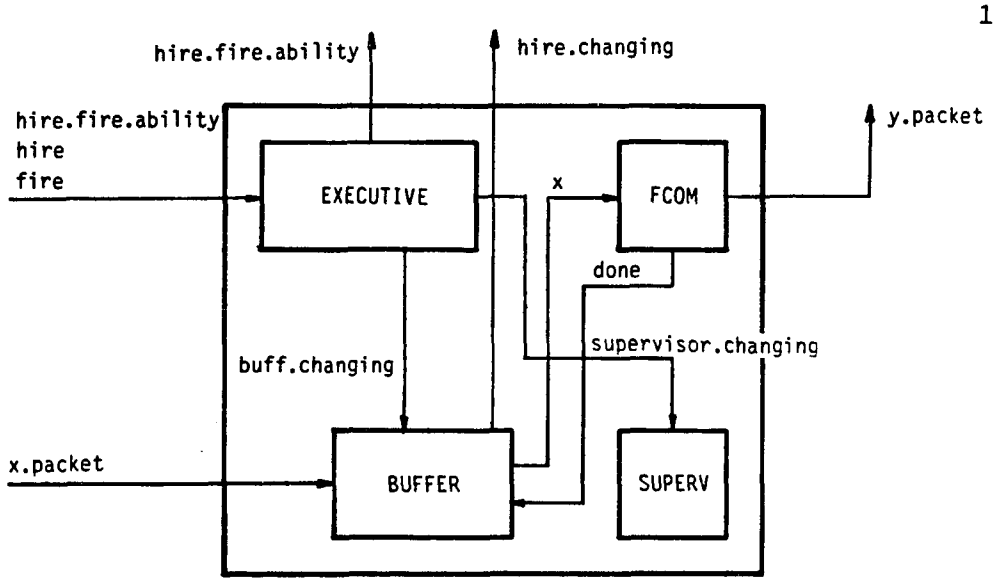
The inputs and outputs of the FCOM are:

External Outputs: y

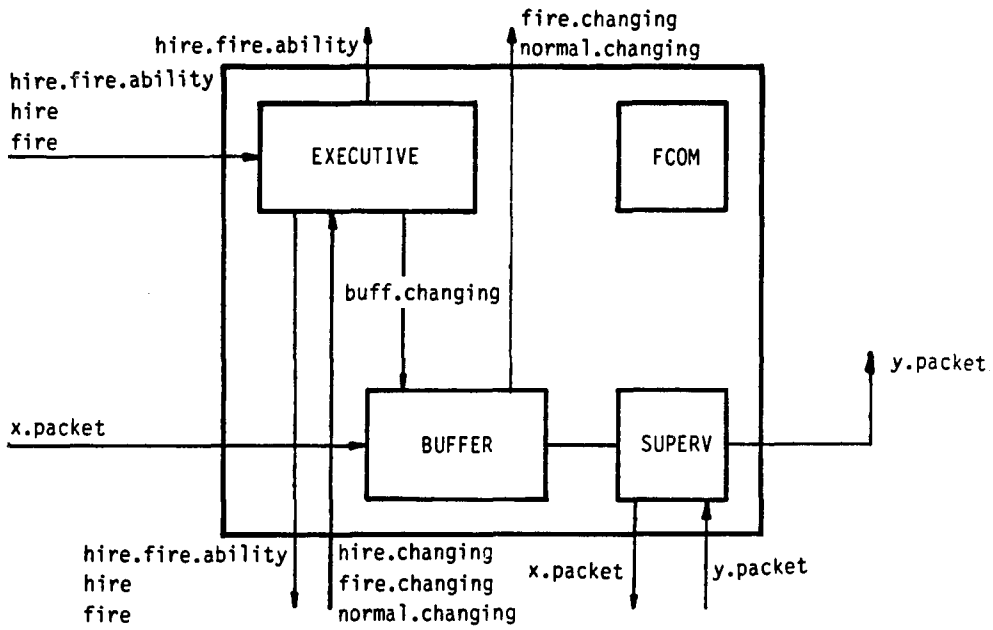
Internal Inputs: x

Internal Outputs: done

The FCOM is only active at the terminal leaves, the "f-computer" type fp's, of the tree structure with the only functional responsibility, processing the packets. Whenever the FCOM receives a packet x from the BUFFER, it determines the computation time T of the packet according to the triple (l, u, j) and, then, processes the packet with T units. Once the processing is over, it sends the problem solution, packet y, back to the SUPERVISOR of the next higher level through a CHANNEL module. In addition, a "done" signal is sent to the BUFFER to inform it that the FCOM has finished the current problem and waits for the next packet from the BUFFER (Figure 2.6 (i)).



(i)



(ii)

Figure 2.6 Internal coupling for (i) an "f-computer," and (ii) a "coordinator"

2.3.2 Buffer Module

The BUFFER module has serval inputs and outputs:

External Inputs: x
buff.flushed
External Outputs: set.flushed.id
hire.changing
fire.changing
normal.changing
Internal Inputs: done
buff.changing(work-type)
Internal Outputs: x

For "f-computers", the main function of this module is to hold all the packets x from its next higher level in a packet queue. Whenever BUFFER receives a "done" signal from the FCOM, it removes the first packet in the queue and sends it to the FCOM (Figure 2.6 (i)). For the "coordinators", this module simply sends the packets, x, to the SUPERVISOR (Figure 2.6 (ii)).

The rest of the signals are related to the "hire/fire" operation. The "buff.changing" signal is from the EXECUTIVE to indicate that the "work-type" of this fp is changing. If it is going to perform a "hire" operation, a "hire.changing" is sent to the next higher level to acknowledge its parent that it has "hired" some fp's. If the fp is carrying out a "fire" operation, a "fire.changing" is sent to its next higher level to indicate it is "firing" its

children; this fp is entering into "changing" mode and its children are, then, set into "inactive" "work-type". When both its children finish all the packets in their BUFFER's queues and are released from the system, a "normal.changing" signal is sent to the next higher level to indicate that it has been recovered from "changing" mode to "normal" mode.

In a "fire" operation, the "f-parent" issues the "set.flushed.id" signal to set its children's "flushed.id" fields which are used to check the status of the fp's to be released. If an fp performs a "fire" operation without receiving "flushed.id" in advance, an error will be reported. The signal "buff.flushed" from the next lower level is used to inform the BUFFER that its child has finished processing all packets in its queue and is ready to be removed from the system.

2.3.3 Supervisory Module

The SUPERVISOR module has I/O as follows:

External Inputs: y
External Outputs: x
y
Internal Inputs: x
changing(coordinator-type)

This module is only active for the "coordinator" type fp's. Its main function is to pass the packets x from the BUFFER to its next lower level according to its

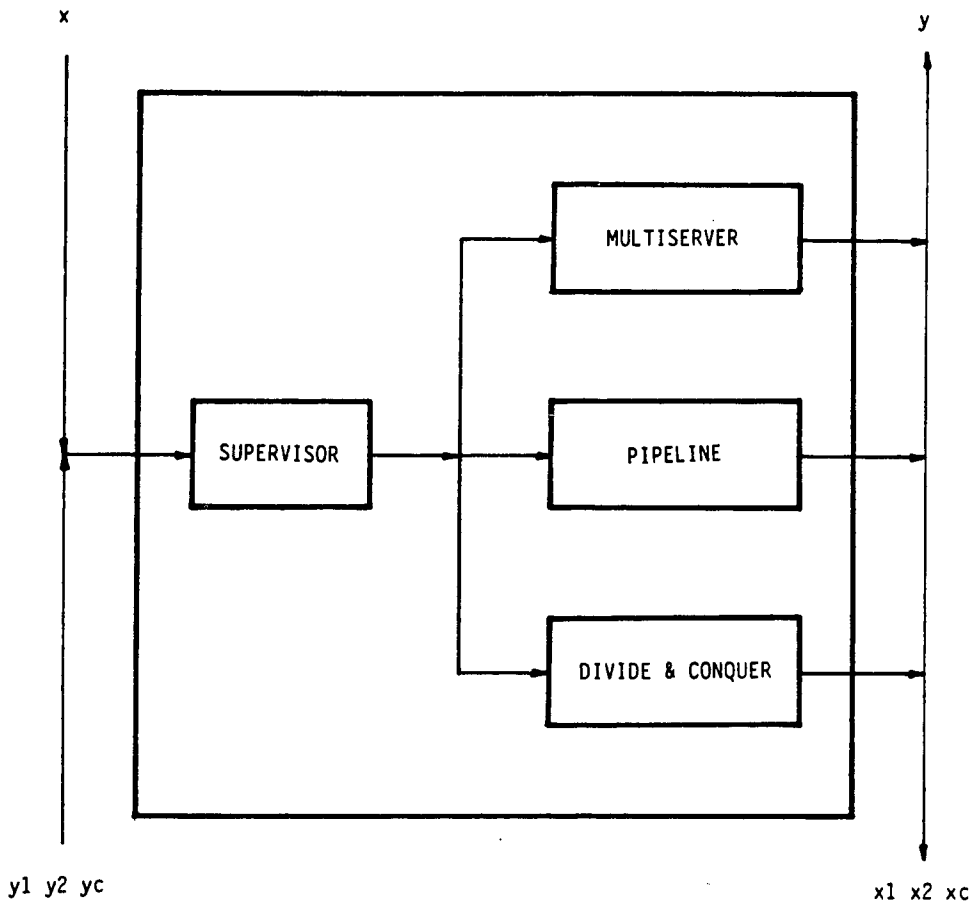


Figure 2.7 Supervisory module and the coupling of "supervisor" and "coordinators"

"coordinator-type" (Figure 2.6 (ii) and 2.7). Similarly, when it receives partial solutions yi (i=1, 2, c) from the next lower level, it will appropriately sends the solutions to the next higher level. When an fp is "hiring", a "supervisor.changing" signal is sent to the SUPERVISOR, which means that change of "coordinator-type" is to be done (Figure 2.6 (ii)).

2.3.4 Executive Module

The EXECUTIVE module has several I/O signals:

External Inputs: hire
fire
hire.fire.ability
hire.changing
fire.changing
normal.changing

External Outputs: hire
fire
hire.fire.ability

Internal Outputs: changing(work-type)
changing(coordinator-type)

The EXECUTIVE is responsible to generate or to accept the signals related to the "hire/fire" operations. The following sections describe the "hire" and the "fire" signals as well as the intervening signals.

2.3.4.1 Hire Signal

Depending on the "hire" generation policy, a "hire" signal can be self generated or comes from the EXECUTIVE of the next higher level. When an "f-computer" receives a "hire" signal, it consults with the "accept.hire.fcom" policy, which is a criterion for determining whether to accept this signal or not. If the policy returns an affirmative answer, the "f-computer" sets itself into "changing" mode because it is changing its "work-type" from "f-computer" to "coordinator". As soon as the FCOM of this fp finishes processing its current packet, all the packets in the BUFFER's queue are passed downward to its new created children. The "supervisor.changing(coordinator-type)" signal is sent to the SUPERVISOR to cause it to become a "coordinator" and the "buff.change(work-type)" signal is sent to the BUFFER informing it of the new "work-type". Also, a "hire.changing" signal is sent to the next higher level to indicate that it's "fp.status" field has changed from "f-computer" to "f-parent".

The "hire" signal is propagated by a "coordinator" to the next lower level according to the "hire.transmission" policy, which determines how and to whom to transmit the "hire" signal.

2.3.4.2 Fire Signal

Based on the "fire" generation policy, this signal can be self generated or received from the EXECUTIVE of the next higher level. When an "f-computer" receives this signal, a "buff.changing(work-type)" is sent to the BUFFER to set it into "inactive"; that is, it will be removed from the system as soon as it finishes all the packets in the BUFFER's queue.

A "coordinator" consults the "fire.transmission" policy to propagate the signal to the next lower level. In the case of "f-parent" and an affirmative answer from the "accept.fire.coord" policy, this fp is set into "changing" mode and sends a "fire.changing" signal to its parent to indicate that its "fp.status" field has changed from "f-parent" to "f-computer". For the "multiserver" and the "divide-and-conquer" coordinators, two "fire" signals are sent to the left and the right "f-computer" respectively. For the "divide-and-conquer", the "fire" signal is sent to the "compiler" after both its left and right children have been released. For the "pipeline" case, only one fire signal is sent to the left child. The second "fire" signal will be sent to the right child after the release of the left child. When it is acknowledged that all of its children has released, the fp is set back to "normal" mode and a "normal.changing" signal is sent to its parent to report that it has returned to "normal" mode.

2.3.4.3 Hire-Fire-Ability Signal

The signal, "hire.fire.ability", is used to initiate the generation of the "hire" and the "fire" signals. Upon receiving this signal, the fp consults with the the "hire.gen.policy" and the "fire.gen.policy". If the answer from either these policies is affirmative, the related signal will be periodically generated from this fp. In a "hire" operation, the hiring node sends two "hire.fire.ability" signals to its new children and also performs a "hire.fire.ability" test itself. For the "fire" operation, the firing node, the "f-parent", sends one "hire.fire.ability" signal to its parent and it will receive a "hire.fire.ability" signal from its children.

2.3.5 Channel Module

The CHANNEL module is an unidirectional transmission medium with infinite capacity, i.e. an unlimited number of messages can be transmitted at one time [6]. Unlike the flexible processors, the CHANNEL is not considered as a resource to be allocated and deallocated. The CHANNEL only describes the communication characteristics between fp's. The communication delay is the main concern and it includes the time delay of preparing the transmission at the source fp, message transmitting in the channel, and receiving at the destination fp. All the Messages transmitted between

fp's must go through the CHANNEL and suffer a constant delay.

Thus in our CHANNEL module, three assumptions are made:

- Unidirectionality: messages can be transmitted in only one direction.
- Infinite capacity: an unlimited number of messages can be transmitted at the same time.
- Constant delay: all the messages are delayed by a constant interval.

The CHANNEL consists of a first-in-first-out infinite capacity queue which holds all the messages being transmitted (Figure 2.8). When the CHANNEL receives a message, it schedules a process in "communication delay" time units for this message and puts this message into the queue. When the "communication delay" time units elapse, the process is activated, the transferred message is removed from the CHANNEL's queue and a related function at the destination fp is invoked. The net effect is that a message, i.e. a function call, through the CHANNEL is delayed.

The CHANNEL is able to handle the following messages:

- problem packet: x
- solution packets: y1, y2, yc
- "hire" and "fire" signals
- "hire.fire.abiltity" signal

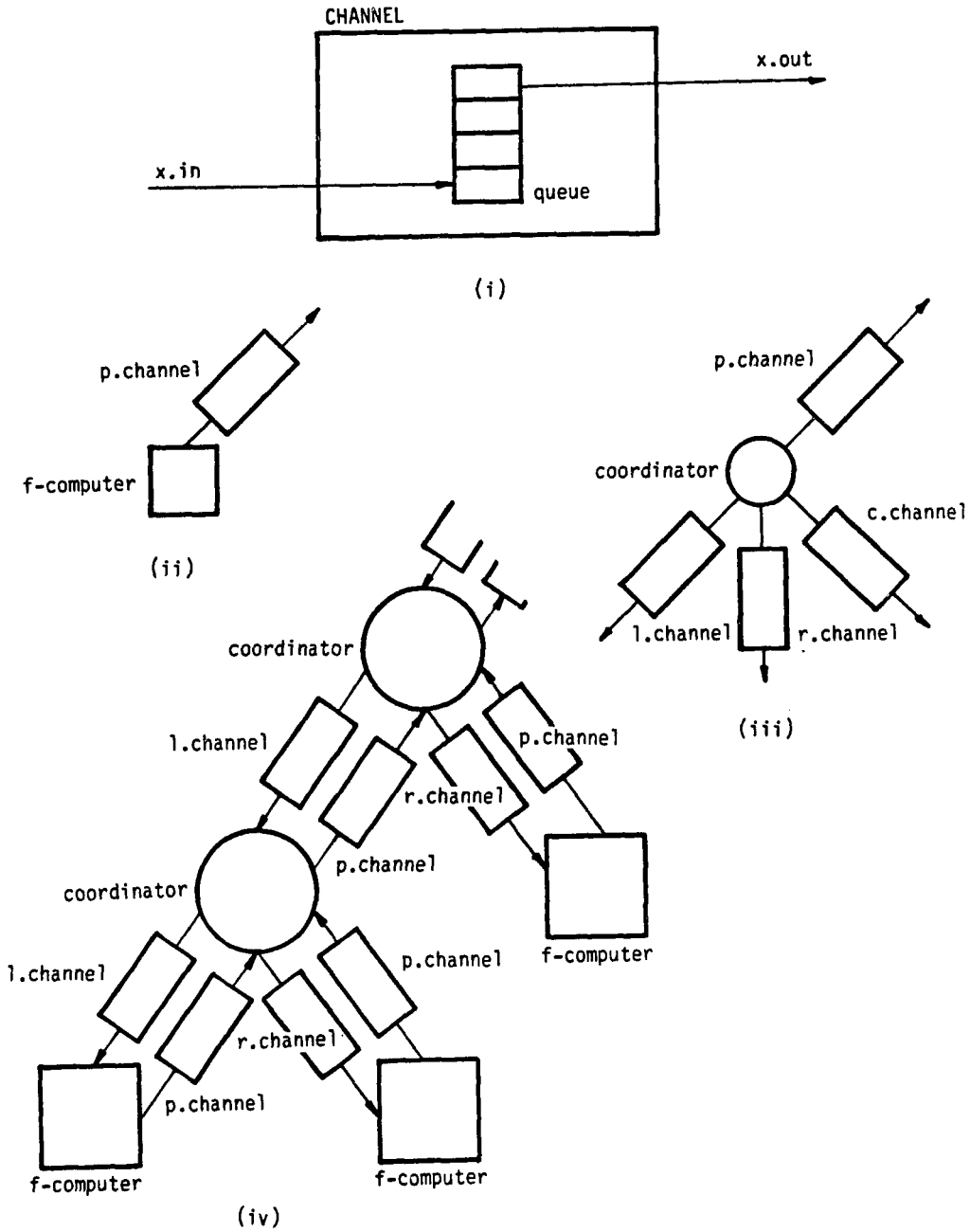


Figure 2.8 (i) A "channel" module. (ii) an "f-computer." (iii) an "coordinator." (iv) the connections of "f-computers" and "coordinators"

. state transition signals: "hire.changing", "fire.changing", "normal.changing"
. "set.flushed.id" signal
. "buff.flushed" signal

There are two CHANNELs between any two fp's; the CHANNEL transmitting messages downward represents the transmission from a parent fp to its child, and the other one indicates the transmission of the reverse direction. A "coordinator" may have up to four CHANNELs, one to its parent if exists, two to its left and right children, and, if the case of "divide'n'conquer", one to its compiler child (Figure 2.8 (iii)). An "f-computer" or a "compiler" only has one channel toward its parent (Figure 2.8 (ii)). Figure 2.8 (iv) shows the CHANNEL connection between the "coordinators" and the "f-computers".

2.3.6 An Example -- Hierarchical Scheme of Flexible Processor

This section gives an example to illustrate the hierarchical organization of the flexible processors (Figure 2.9). The hierarchy consists the first level -- a "coordinator" and the second level -- an "f-computer" and a "coordinator". Any message transmission between fp's goes through the CHANNEL, which is not shown in the figure for simplicity.

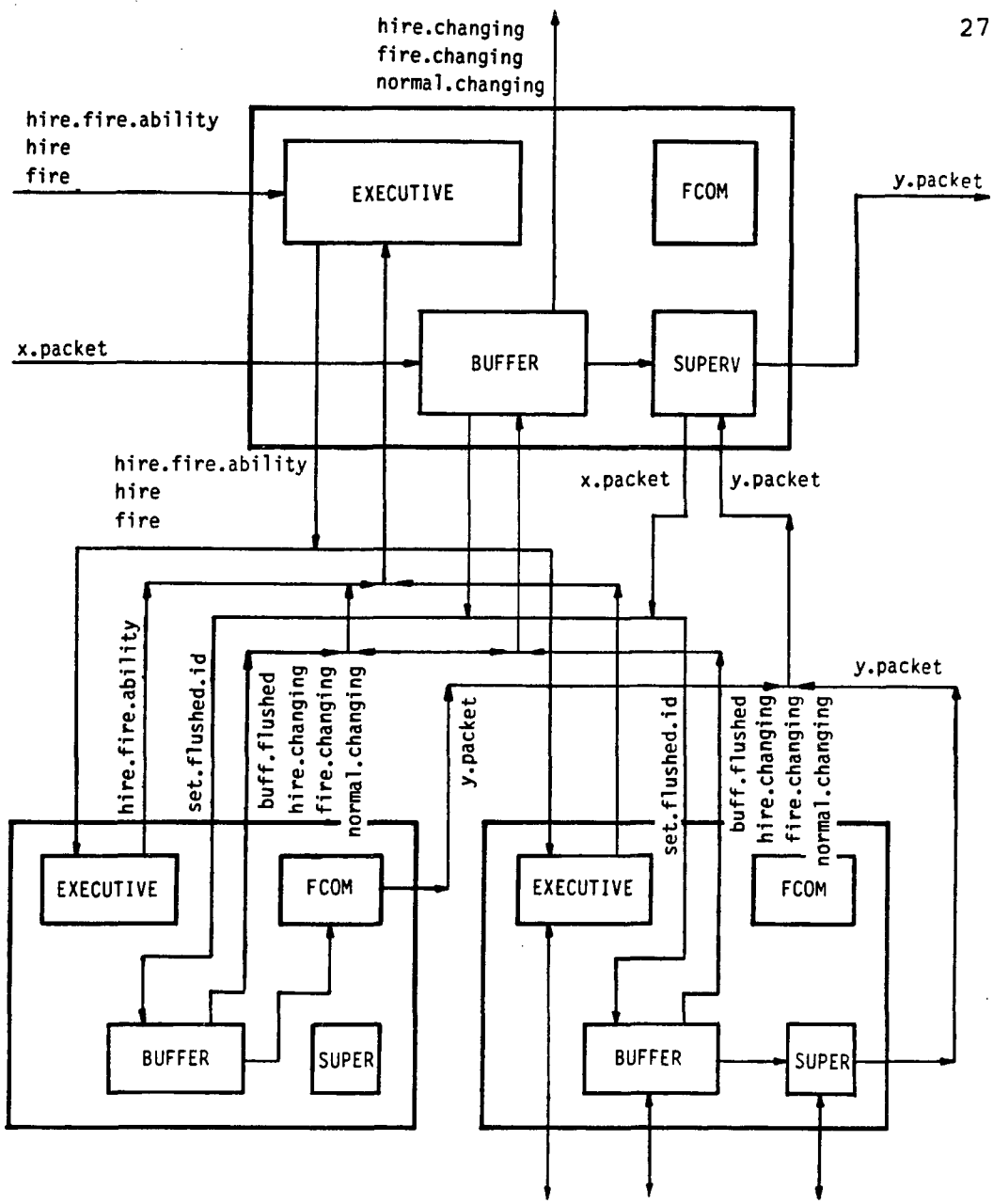


Figure 2.9 External coupling of flexible processors

The packets are sent to the first level coordinator from the outside world or the next higher level "coordinator". The packets are passed from the BUFFER to the SUPERVISOR, who determines the appropriate children to receive the packets, and, then through the CHANNEL, to the BUFFER of the second level. For the left child, the BUFFER sends the packets into its queue and send to the FCOM one packet at a time. The done packets are sent upward to the SUPERVISOR of the first level, and, again, transmitted upward until the packets reach the root and leave the system. For the right child of the second level, the packets is sent to the SUPERVISOR and routed downward appropriately.

The "hire", the "fire", and the "hire.fire.ability" are always routed to the EXECUTIVE. The "coordinator" of the first level may generate the "hire/fire" itself according to the generation policies or receive them from the next higher level. By consulting the "hire/fire" transmission policy, the "hire/fire" are transmitted to the second level appropriately. For the left child, it consults a hire accept policy when receiving a "hire" signal and creates new children if the answer from the policy is affirmative. If it receives a "fire" signal, it is set "inactive" and released from the system when all the packets in the BUFFER's queue are done. Similarly to the first level, the right child consults the transmission policy to transmit "hire/fire" appropriately. In the case of the "f-parent", it consults

the fire accept policy, and, if the answer is ratifying, the "fire" signal is transmitted downward properly according to the coordinator type. The fp receiving the "hire.fire.ability" consults the "hire/fire" generation policy, and, if returned with confirming answer, a "hire.test" and/or a "fire.test" is scheduled.

2.4 Flexible Processor's Status

In "normal" mode, an fp can be an "f-computer" or a "coordinator". An fp in the "changing" mode indicates that a transformation of its "work-type" is to be done. The transformation occurs when an fp receives a "hire/fire" signal and gets an affirmative from the related policy. While "hiring", the "f-computer" becomes a "coordinator" by creating two new children. In the "firing" case, an "f-parent" fp becomes an "f-computer" by firing its children and an "f-computer" changes to "inactive". The "changing" mode is set in these operations, and details are described in the following sections.

2.4.1 Normal Mode

In the "normal" mode, the "f-computer" fp's send the packets to their BUFFERs which direct packets to the FCOM one at a time, and, in the case of the "coordinator" fp's, the packets are sent to the SUPERVISORs which propagate them to the next lower level.

The "hire", the "fire", and the "hire.fire.ability" signals are always routed to the EXECUTIVE. In the case of "hiring" and "f-computer", if a confirming answer is obtained from the accept policy, it follows these steps:

. sets to the new coordinator type which is carried by the "hire" signal by sending "coord.change" to the SUPERVISOR.
. sends a "hire.changing" signal to its parent.
. enters "changing" mode if the BUFFER's queue is not empty or the FCOM is busy.
. allocates new nodes.
. performs a "hire.fire.ability" test and also sends this signal to its child.

A "coordinator" transmits the "hire/fire" signals to the next lower level by consulting the transmission policy. In the case of "f-parent" and receiving a "fire" signal, if there is an affirmative answer from the fire accept policy, it follows these steps:

. sets its "work-type" to "f-computer" and sends a "fire.changing" signal to its parent.
. enters "changing" mode.
. sets children's "flushed.id" and sends "fire" signals to the children.
. waits until receiving all its children "buff.flushed" and set itself back to "normal" mode and sends a "normal.changing" to it parent.

The "pipeline" is a special case here. The "fire" signal is sent to the left child first, and, until the left is released from the system, the second "fire" signal is sent to the right child.

2.4.2 Changing Mode

An fp can be set into "changing" mode when it carries out either a "fire" or a "hire" operation.

In the case of "hiring" and "coordinator", in which the "f-computer" fp "hires" new children and becomes a "coordinator", there are two possibilities for the fp being in "changing" mode:

. the FCOM module is busy, or

. the BUFFER's queue is not empty

At this moment, the incoming packets are pushed into the BUFFER's queue temporarily. In the first case, the fp is waiting for the "done" signal from the FCOM. Upon receiving the "done", no packet is sent to the FCOM any more and all the packets in the BUFFER's queue are transmitted downward through the SUPERVISOR and the CHANNEL to the next lower level according to its "coordinator-type". Now the fp is in the second case; there are packets in its queue. When all the packets in the BUFFER's queue are cleared, the fp is set back to the "normal" mode.

In the case of "firing" and "inactive", in which an "f-computer" fp receives a "fire" signal and is ready to be

released from the system, the fp is in "changing" mode for the same reasons as above:

. the FCOM module is busy, or

. the BUFFER's queue is not empty

When a packet in the FCOM is done, the first packet in the BUFFER's queue will be removed and sent to the FCOM. Until all the packets are done, it sends the "buff.flushed" to its parent and, following this, is released from the system. It is not necessary to set back "normal" mode again.

In the case of "firing" and "f-computer", in which an "f-parent" fp is "firing" its children and becomes an "f-computer", the fp is in "changing" mode because of waiting for its children's release from the system. As soon as all its children are released, it is set into "normal" mode and initiates the operation as an "f-computer". A "normal.changing" signal is also sent to its parent.

2.5 Flexible Processor's Structure State

The system configuration of the adaptive computer is changing over time according to the work load of the system. Each fp in the system has different tasks to accomplish. For instance, an ordinary "coordinator" only transmits the "fire" signal to its children but an "f-parent" "coordinator" has to determine whether to accept the "fire" signal or not. We shall represent these changes by considering that each fp is in a different state, which

signifies its relationship to the others. This state concept was introduced in order to facilitate modular design of the system. It is applied when:

. transmitting a "fire" signal

. accepting a "fire" signal

However, in the "hire" signal handling, it is easier; the "coordinator" transmits it to one of its children and the "f-computer" determines whether to accept or to reject it. Thus, the state concept is not used in this case.

For example, when a "coordinator" receives a "fire" signal from its parent, it may transmit this "fire" signal to one of its "coordinator" children according to the transmission policy or it may accept and pass this "fire" signal to all its children. The problem is that how can an fp know what its children's types are: "coordinator" or "f-computer"? A solution is that every fp must be able to keep track of the states of its children. An fp may change its "work-type" as soon as it receives "hire/fire" signals and carries out the related operation. Its state also changes because it "hires" new children or "fires" its children. Since its relationship with other fp's changes, whenever an fp's state changes, it must also inform its parent of its transition; its parent's state also changes.

2.5.1 State Representation

The state of a flexible processor is described by its "fp.status" field, the number of its sub-type children, and the number of the "normal.changing" signals to be received. The state can be expressed as a triple:

(F/N/C)

where:

F: "fp.status" of fp with range: {"parent", "f-parent", "f-computer", "inactive" } or we also use numbers to express this field:

- 3 - "parent"
- 2 - "f-parent"
- 1 - "f-computer"
- 0 - "inactive"

N: number of sub-type children: {3, 2, 1, 0}

- 3 - both right and left sub-type children*.
- 2 - only right sub-type child.
- 1 - only left sub-type child.
- 0 - no sub-type child.

* An fp's sub-type children are the fp's of their "fp.status" one level lower than it (their "F" field value minus one). For example, a "parent's" sub-type children are of type "f-parents". An "f-parent's" sub-type children are of type "f-computers", and so on.

C: the number of "normal-changing" signals to be received in case of receiving the "fire-changing" signals from its sub-type children: {3, 2, 1, x}

3 - signals from both children to be received; none of the children has sent this signal.

2 - signal from right child to be received; left child has already sent this signal.

1 - signal from left child to be received; right child has already sent this signal.

x - don't care condition; this field is not used in some states.

In this representation, for example, an "f-computer" is indicated by the triple (1/0/x) in which its "F" field is "f-computer" and the number of sub-type children is zero because it is a terminal leaf and the "C" field is not used in this state. An "1-parent" is represented by the triple (3/1/1) which shows it is "parent" and expects to receive one "normal.changing" signal from its left child, its only sub-type child.

2.5.2 Flexible Processor's States

Before going on the examples, some terminology must be explained first in order to understand better:

. inactive (x/x/x): is a terminal leaf which receives the "fire" signal and is ready to be removed from the system as soon as it finishes all packets in the BUFFER's queue.

- f-computer (1/0/x): a functional undecomposable terminal leaf (Node "A" in Figure 2.10 (i))
- f-parent (2/3/x): a node with two "normal" mode "f-computer" children (Node "A" in Figure 2.10 (ii)).
- pseudo-parent (3/0/3): a node with two "changing" mode "f-computer" children; its children are under the "fire" operations.
- l-pseudo-parent (3/0/2): a node with its left child as a "changing" mode "f-computer" and its right child as a "normal" mode "f-computer"; its right child has finished its "fire" operation and its left child is still under a "fire" operation.
- r-pseudo-parent (3/0/1): a node with its right child as a changing mode "f-computer" and its left child as a "normal" mode "f-computer"; its left child has finished its "fire" operation and its right child is still under a "fire" operation (Node "A" in Figure 2.12 (iii)).
- parent (3/3/3): a node with all its children of type "parent" or "f-parent" (Node "A" in Figure 2.10 (iV) and Figure 2.12 (iii)).
- l-parent (3/1/1): a node with left child of type "f-parent" and right child of type "normal" mode "f-computer" (Node "A" in Figure 2.10 (iii)).
- r-parent (3/2/2): a node with right child of type "f-parent" and left child of type "normal" mode "f-computer" (Node "A" in Figure 2.12 (iii))

2.5.3 State Transitions

If the work load of an fp satisfies some criteria (which are defined in the "hire accept policy" and the "fire accept policy") at the time of receiving a "hire" or a "fire" signal, a "hire" or a "fire" operation will be initiated. After the "hiring" or the "firing" operation, the fp's state changes, and so does its parent's. Figure 2.11. and 2.13 show state transition diagrams that were used to manage the "hire" and the "fire" operations. The signal and fp with prefix "r" and "l" means "right" and "left" respectively. For instance, "r.hire.changing" stands for a signal "hire.changing" from the "right" child and "l-parent" represents a "parent" fp with a "left" sub-type child. The prefix "to" of an fp means that the fp is in a transition state and is expecting some more signals to complete the state transition. For example, a "to-r-parent" fp has received an "l.fire.changing" and is waiting for an "l.normal.changing" to complete its state transition from "parent" to "r-parent". In addition, we assume the "root-root" generation policy in these examples. We will discuss two examples to illustrate the operations of the state transition schemes. In these examples, the quoted capitalized letters such as "A" and "B" indicates fp's and the quoted digits such as "1" and "2" stands for the signals transmitted and their sequential order.

2.5.3.1 Hire Operation and State Transition

Performing a "hire" operation will change the state of an fp and its parent's. A complete state transition for the "hire" operation is shown in Figure 2.11. The first example demonstrates how the system grows from a single fp to a tree by keeping on "hiring" fp's:

a) Initially, "A", an "f-computer" (1/0/x), receives packets from the outside world (Figure 2.10 (i)).

b) "A" generates a "hire" signal, "1", and carries out a "hire" operation according to a confirming answer by consulting the "hire accept policy". After "hiring" "B" and "C", "A" becomes an "f-parent" (2/3/x) (Figure 2.10 (ii) and 2.11).

c) Later on, a "hire" signal "1" is generated by "A" and transmitted to "B" (1/0/x) according to the "hire" transmission policy (Figure 2.10 (ii)). Assume that "B" accepts the "hire" signal. Step (b) is repeated.

d) After "B's" "hiring", a "hire changing", "2", indicating that "B's" state has changed from "f-computer" to "f-parent" (2/3/x) is sent to "A". When "A" receives this signal, it switches its state from "f-parent" to "1-parent" (3/1/1) (Figure 2.10 (iii) and 2.11).

e) Similarly, a "hire" signal, "3", is generated and routed to "C" (1/0/x). Assume that "C" accepts this "hire" and step (b) is repeated. Again, a "hire.changing", "4", from

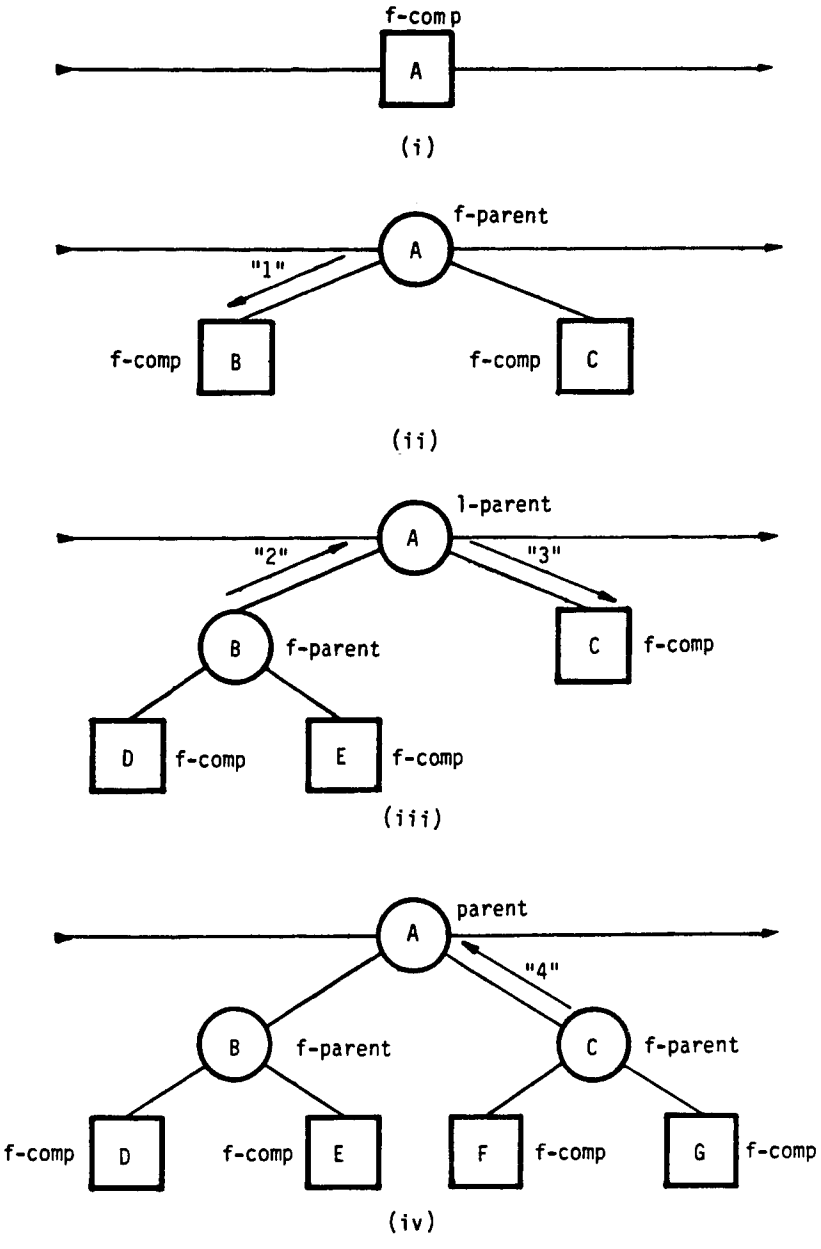


Figure 2.10 System grows from a single processor to a tree: (i) an "f-computer." (ii) an "f-parent." (iii) an "1-parent." (iv) a "parent"

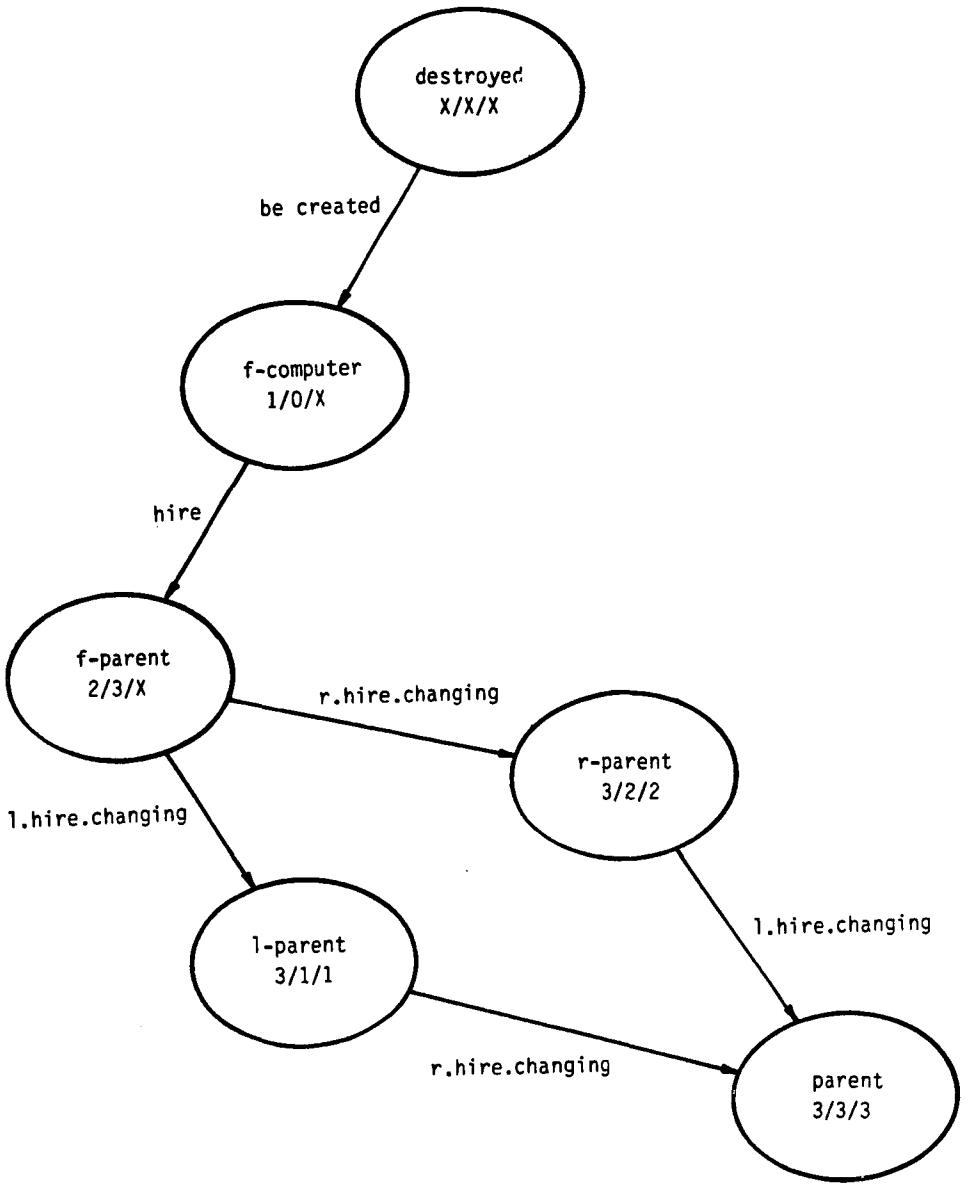


Figure 2.11 State transition diagram at "hiring"

"C" is sent to "A", and "A" switches its state from "1-parent" to "parent" (3/3/3) (Figure 2.10 (iv) and 2.11).

2.5.3.2 Fire Operation and State Transition

Processing of a "fire" operation will also change an fp's state and its parent's. A complete state transition for the "fire" operation is shown in Figure 2.13. The second example illustrates how the transformation from a tree to a single fp is done by continual "firing" of fp's:

a) Consider a "parent" "A" (3/3/3) with its two "f-parent" children (Figure 2.12(i)). A "fire" signal, "1", is transmitted from "A" to "B" according to the "fire transmission policy", which determines which child should receive this "fire" signal (Figure 2.12(i)).

b) Assume that "B" accepts the "fire" signal by consulting with the "fire acceptance policy" and getting a confirming answer. Immediately, "B" is set into "changing" mode and sends a "fire.changing", "2", to its parent "A" to indicate that its state changes from "f-parent" (2/3/x) to "to-f-computer" (1/3/x). Upon receiving this changing signal, "A" also switches its state from "parent" (3/3/3) to "to-r-parent" (3/2/3). Then, "B" sends two "set.flushed.id" signals and two "fire" signals its children, "D" and "E", which will be changed to "inactive" if the BUFFER's queues of these

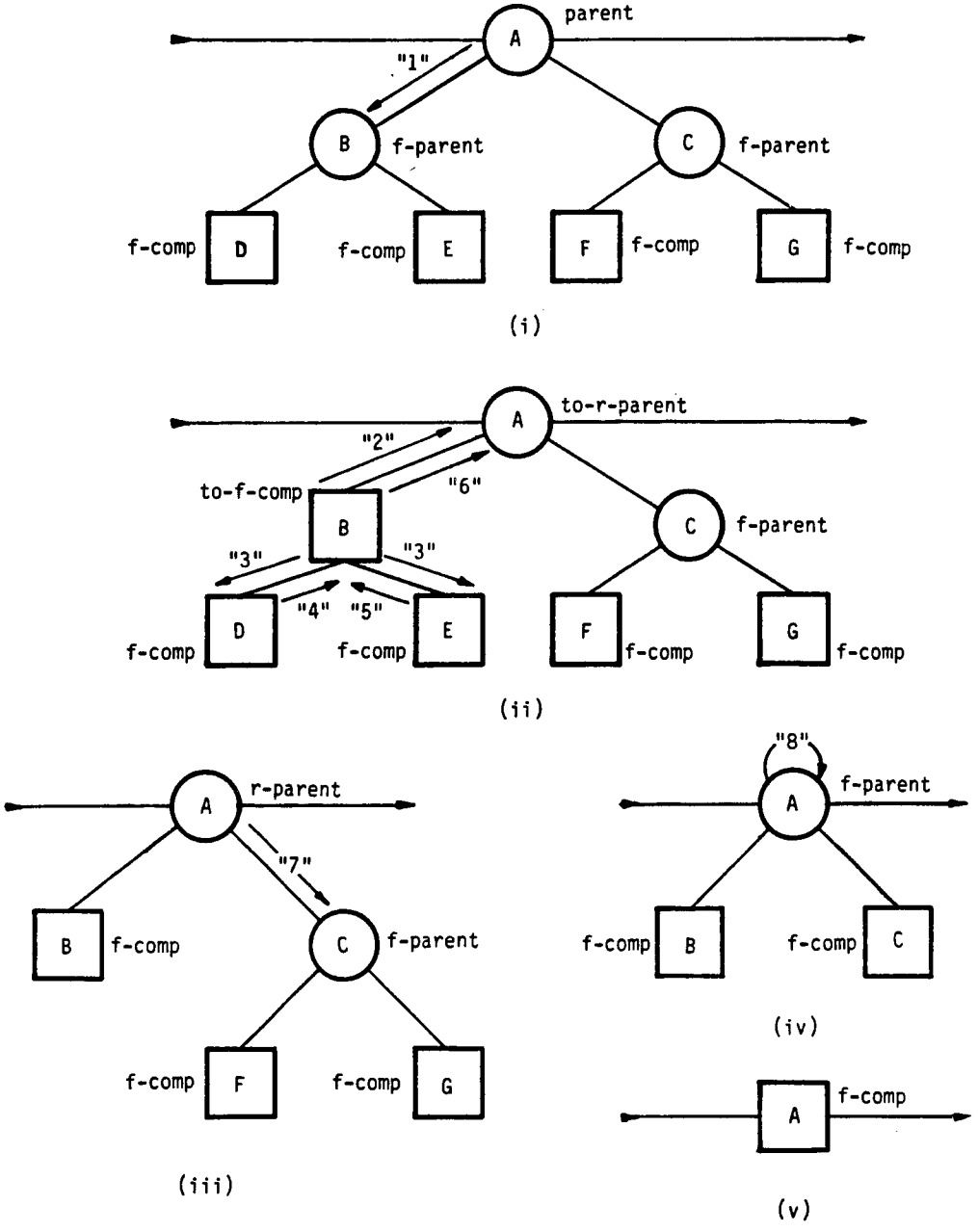


Figure 2.12 System degrades from a tree to a single processor: (i) a "parent". (ii) a "to-r-parent". (iii) an "r-parent". (iv) an "f-parent". (v) an "f-computer"

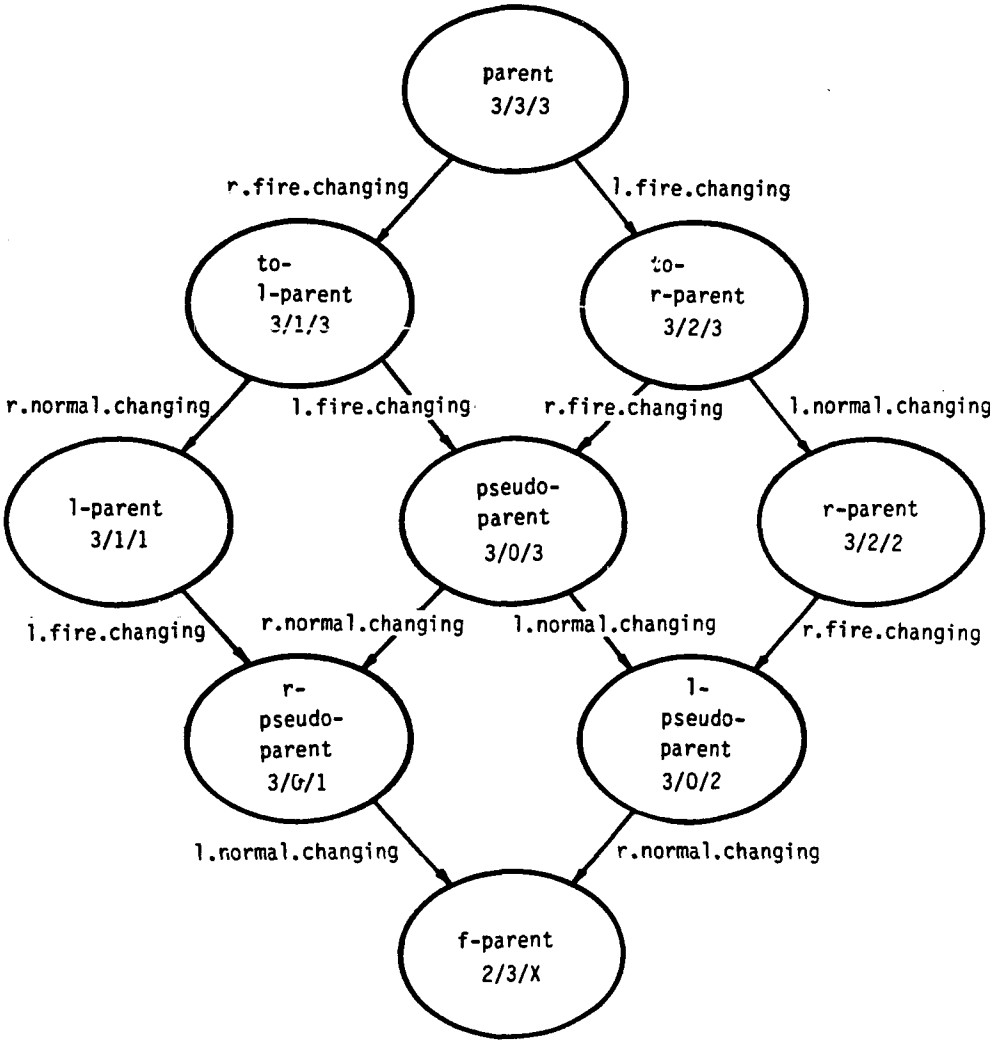


Figure 2.13 State transition diagram at "firing" (To be continued)

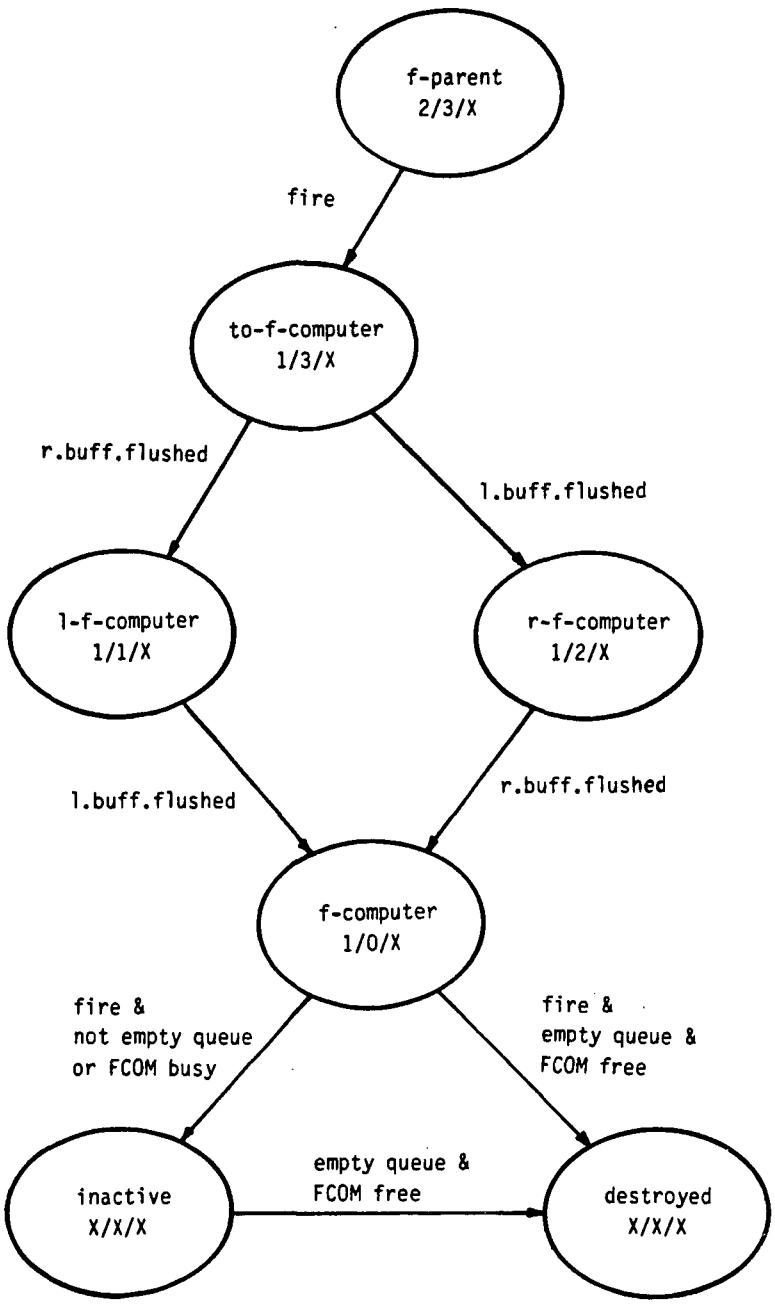


Figure 2.13 (Continued)

fp's are not empty or their FCOM's are busy (Figure 2.12 (ii) and 2.13).

c) When "D" (x/x/x) finishes processing all the packets in its BUFFER's queue, it sends a "buff.flushed" signal, "4", to "B" to indicate that it has finished all the packets and is ready to be released from the system. When "B" (1/3/x)receives these signals, it knows that its left child is removed and, therefore, becomes an "r-f-computer" (1/2/x) (Figure 2.12 (ii) and 2.13).

d) Similarly, "E" (x/x/x) is released from the system in the same way as in step (c). The "buff.changing" signal, "5", is sent to "B" (1/2/x), and "B" changes its state to "f-computer" (1/0/x). At this moment, a "normal.changing" signal, "6" is sent to "A" (3/2/3) to acknowledge that "B" has recovered from "changing" mode to "normal" mode. (Figure 2.12 (ii)) By receiving this signal, "A" becomes an "r-parent" (3/2/2) (Figure 2.12 (iii) and 2.13).

e) A "fire" signal, "7", is routed from "A" to "C" according to the "fire transmission policy". Note that, in this case of "fire" signal transmission, a "fire" signal will never be transmitted directly to an "f-computer", "B", and must be first intercepted by an "f-parent" (Figure 2.12 (iv)).

f) Assume that "C" accepts this "fire" signal. Step (b), (c), and (d) are repeated. "A" become an "f-parent"

(2/3/x) with two "f-computer" children, "B" and "C" (Figure 2.12 (iv) and 2.13).

g) A "fire" signal, "8", is generated and accepted by "A". Step (b), (c), and (d) are repeated again. "A" becomes an "f-computer" (1/0/x) (Figure 2.12 (v) and 2.13).

CHAPTER 3

FORMAL MODEL DESCRIPTIONS

The data attributes of the packet, the communication channel, the flexible processor, and the coupling of flexible processor, in our simulation model are described in pseudo codes in this chapter. Additions to, the modifications of Linos' model [4] are indicated by an asterisk, "*", at the end of the statement and the comments of statements follow a semi-colon, ";".

3.1 Data Types

There are three major data structures which are SIMSCRIPT entities -- the packet, the flexible processor, and the channel, used in our model.

3.1.1 Packet

```
(l, u, j, found, parent.packet, l.packet, r.packet,
packet.id)
i, u, j  integer,
found    boolean,
packet-list with range packet.range
```

3.1.2 Flexible Processor

```
work.type w/r {f-computer, co-ordinator, inactive}
co-ordinator-type w/r
              {co-ordinator of multiserver,
               co-ordinator of pipeline,
               co-ordinator of divide'n'conquer}
fn-type w/r {full, lowerhalf, upperhalf, compiler}
fp.status w/r {parent, f-parent, f-computer, inactive}*
changing w/r boolean,
parent       node,
lchild       node,
rchild       node,
compiler     node,
pchannel     channel,* ;pointer to the channel to parent
lchannel     channel,* ;pointer to the channel to lchild
rchannel     channel,* ;pointer to the channel to rchild
cchannel     channel,* ;pointer to the channel to compiler
hire.sched   event,*   ;pointer to the "hire.test" event
fire.sched   event,*   ;pointer to the "fire.test" event
leaf         integer,
sendee.free  integer,
length1      integer,
length2      integer,
```

```
id.side          integer,* ;number identifying the position
                           ;related to its parent:
                           ;1 = left, 2 = right, 3 = compiler
num.child        integer,* ;the number of sub-type children
num.change       integer,* ;the number of "normal.changing" to
                           ;be received
register         array[3],
flushed.id       integer,
packets.waiting  integer,
queue            packet.list
```

3.1.3 Channel*

```
channel packet:
ch.x          packet,    ;pointer to channel packet
ch.new.type   text,      ;attribute carrying the text variable
                         ;of the transmitted message
ch.oper       integer,   ;message type
ch.done       boolean,   ;channel message status
capacity      channel.packet.list
channel:
dest.node     node,       ;pointer to destination fp
busy          boolean,    ;channel status
current.pac   packet,     ;pointer to current packet
process.ptr   event       ;pointer to current event
```

3.2 Coupling of Flexible Processor

The coupling of processors can classified into the "external" and the "internal" coupling. The "external" coupling represents the linkages between processors and they must go through "channels". The "internal" coupling is the connections of "modules" within a processor and they do not go through the "channel".

3.2.1 External Coupling

The connections described in this section are going through the CHANNELs but only the signal sources and the destinations are shown for simplicity.

input [flushed1, flushed2, flushedc, x] =

BUFFER.[flushed1, flushed2, flushedc, x]

input [hire, fire, hire-fire-ability, hire-changing, fire-changing, normal-changing] =

EXECUTIVE.[hire, fire, hire-fire.ability, hire-changing, fire-changing, normal-changing]*

input [y1, y2, yc] = SUPERVISOR.[y1, y2, yc]

output [hire, fire, hire-fire-ability] =

EXECUTIVE.[hire, fire, hire-fire-ability]

output [set-flushed-id, flushed, hire-changing, fire-changing, normal-changing] =

BUFFER.[set-flushed-id, flushed, hire-changing, fire-changing, normal-changing]*

output [y] = [FCOM.y or SUPERVISOR.y]

output [x1, x2, xc] = SUPERVISOR.[x1, x2, xc]

3.2.2 Internal Coupling

input x = FCOM.x or SUPERVISOR.x

input [done, buff-changing] = BUFFER.[done, buff-changing]

input supervisor-changing = SUPERVISOR.supervisor-changing

output done = FCOM.done

output x = FCOM.x or SUPERVISOR.x

output [buff-changing, supervisor-changing] =

EXECUTIVE.[buff-changing, supervisor-changing]

3.3 Simulation Model Descriptions

In this section, two major modules, the channel and the fp, of our work are explained in pseudo codes, which are consistent with the simulation programs written in SIMSCRIPT II.5.

3.3.1 Channel Module*

```
Inputs:  x
Outputs: y
when receive x
    insert x to queue
    activate process x
    done(x) := False
    if one(queue) then
        goto REPEAT
    else
```

```
        continue
    endif
REPEAT:
    x := first(queue)
    queue := rest(queue)
WAIT:
    wait.until(x.done)
    send x to external
    if queue is not empty
        goto REPEAT
    endif
P:  passivate
end
Process x
    hold(channel.delay)
    done(x) := True
    passivate
```

3.3.2 Flexible Processor Module

3.3.2.1 Executive Module

External Inputs:

- hire-signal(new-type)
- fire-signal
- hire.fire.ability
- hire.changing*
- fire.changing*

```
        normal.changing*
External Outputs:
        hire-signal(new-type)
        fire-signal
        hire.fire.ability
Internal Outputs:
        buff.changing(work.type)
        supervisor.changing(coordinator-type)
Initialize:
       fp.status := "fcom", {"par", "fpar", "fcom", "inac"}*
       num.child := 0*
       num.change := 0*
when receive hire-command(new-type)
if not changing then
     if work-type = "f-computer" and accept-hire.fcomp then
         send change (new-type) to SUPERVISOR
         send change("coordinator") to BUFFER
         create hire.fire.ability(self)
         create new-nodes(new-type)
         send hire.fire.ability.test to CHANNEL(EXEC(new-
nodes))*
         fp.status := "fpar"*
         num.child := 3*
     else
         hire-transmission-policy
     endif
```

```
endif
passivate
end
when receive fire-command
if not changing then
     if work-type = "f-computer" then
         if parent exists then
            send hire.fire.ability to CHANNEL(EXEC(parent))*
         endif
         send change("inactive") to BUFFER
         fp.status := "inac"*
     else
         if work.type="coordinator" and accept.fire.fparent
             if f.parent then
                 send change("f-computer") to BUFFER
                 fp.status := "fcom"*
                 if not root then
                     send hire.fire.ability to
                                     CHANNEL(EXEC(parent))*
                 endif
                 send fire.command to
                                     CHANNEL(EXEC(lchild)))*
                 if coordinator <> "pipeline" then
                 send fire.command to
                                     CHANNEL(EXEC(rchild))*
                 endif
```

```
            else
                if not pseudo.f.parent then
                    fire.transmission.policy
                endif
            endif
        endif
    endif
endif
passivate
end
when receive changing(hire, id)*
if fp.status = "fpar" then
    fp.status := "par"
    num.child := id
    num.change := id
else
    if fp.status = "par" then
        if num.child = 3 then  ;already hire before
            error
        endif
        num.child := id + num.child
        num.change := id + num.change
        if num.child > 3 or num.change > 3 then
            error
        endif
    else
```

```
        error
    endif
endif
passivate
end
when receive changing(fire, id)*
if fp.status = "par" then
    if num.child = 3 - id then ;already fire before
        error
    endif
    num.child = num.child - id
    if num.child < 0 then
        error
    endif
else
    error
endif
passivate
end
when receive changing(normal, id)*
if fp.status = "par" then
    if num.change = 3 - id then ;already received before
        error
    endif
    num.change := num.change - id
```

```
    if num.change = 0 and num.child = 0 then
        fp.status = "fpar"
        num.child = 3
    endif
else
    error
endif
passivate
end
fire.transmission.policy :
if length1 > length2
    if r.parent then
        send fire.command to CHANNEL(EXEC(rchild))*
    else
        if l.parent then
            send fire.command to CHANNEL(EXEC(lchild))*
        endif
    endif
else
    if l.parent then
        send a fire.command to CHANNEL(EXEC(lchild))*
    else
        if r.parent then
            send a fire.command to CHANNEL(EXEC(rchild))*
        endif
    endif
```

```
endif
end
hire.transmission.policy :
if work.type = "coordinator" then
    if coordinator.type="div and conq" or "pipeline" then
        if packets.waiting >= hire.limit then
            send hire.command to CHANNEL(EXEC(lchild))*
            send hire.command to CHANNEL(EXEC(rchild))*
        endif
    else
        if length1 <= length2
            send hire.command to CHANNEL(EXEC(rchild))*
        else
            send hire.command to CHANNEL(EXEC(lchild))*
        endif
    endif
endif
end
hire.fire.ability :
if hire.gen.policy then
    if hire.sched = 0*
        activate hire.generation.process
    else
        if m.ev.s(hire.sched) = 0*
            activate hire.generation.process
        endif
```

```
    endif
endif
if fire.gen.policy then
    if fire.sched = 0*
        activate fire.generation.process
    else
        if m.ev.s(fire.sched) = 0*
            activate fire.generation.process
        endif
    endif
endif
end
hire.generation.process :
START: send hire.command(new.type) to EXECUTIVE
       if hire.gen.policy then
           hold(gen.time)
           goto START
       else
PASSV:     passivate
       endif
       end
fire.generation.process :
START: send fire.command to EXECUTIVE
       if fire.gen.policy then
GEN.DELAY: hold(gen.time)
           goto START
```

```
        else
PASSV:      passivate
        endif
        end
function accept.hire.fcomp :
if time is zero
    accept.hire.fcomp := true
else
    if length(que) >= hire.limit
        accept.hire.fcomp := true
    endif
endif
end
function accept.hire.fpar :
if packets.waiting <= fire.limit then
    accept.fire.fpar := true
endif
end
function hire.gen.policy :
gen.type : {root.type, f.parent.type}
root.type :
            if root then
                hire.gen.policy := true
            endif
f.parent.type :
            if f.parent or (root and f.computer)
```

```
                hire.gen.policy := true
            endif
end
function fire.gen.policy :
gen.type : {root.type, f.parent.type}
root.type :
    if root then
        hire.gen.policy := true
    endif
f.parent.type:
    if f.parent or pseudo.f.parent then
        fire.gen.policy := true
    endif
end
function f.parent :*
if fp.status = "fpar" then
    fpar := true
endif
end
function pseudo.f.parent :*
if fp.status = "par" and num.child = 0
    pseudo.f.parent := true
endif
end
function r.parent :*
if fp.status = "par"
```

```
    if num.child = 3 or num.child = 2
        r.parent := true
    endif
endif
end
function 1.parent :*
if fp.status = "par"
    if num.child = 3 or num.child = 1
        1.parent := true
    endif
endif
end
```

3.3.2.2 Supervisory Module

3.3.2.2.1 Supervisor Module

```
External Inputs:
        y1, y2, yc
External Outputs:
        x, y
Internal Inputs:
        x
        supervisor.changing(coordinator-type)
when receive < x, y1, y2, yc >
    send <x, y1, y', yc> to CO-ORDINATOR(co-ordinator-type)
passivate
```

```
when receive change(new-type)
    co-ordinator-type := new-type
passivate
end
```

3.3.2.2.2 Multiserver

```
Inputs:  x, y1, y2
Outputs: x1, x2, y
when receive x
    increment packets.waiting
    xj := x
    packet.id(xj) := x
    send xj to CHANNEL(BUFFER(child j))*
    increment length.j
passivate
end
when receive yi, i=1, 2
    decrement packets.waiting
    y := parent.packet(yi)
    if not root then
        send y to CHANNEL(SUPERVISOR(parent))*
    else
        destroy y
        notify output
    endif
    decrement length.i
```

```
passivate
```

3.3.2.2.3 Pipeline

```
Inputs:  x, y1, y2
Outputs: x1, x2, y
when receive x
    increment packets.waiting
    x1 := lower(x)
    packet.id(x1) := 1
    send x1 to CHANNEL(BUFFER(lchild))*
    increment length.1
passivate
end
when receive y1
    decrement length.1
    y := parent.packet(y1)
    if found(y1) then
        decrement packets.waiting
        found(y) := found(y1)
        if not root then
            send y to CHANNEL(SUPERVISOR(parent))*
        else
            destroy y
            notify output
        endif
```

```
    else
        x2 := upper(y)
        packet.id(x2) := 2
        send x2 to CHANNEL(BUFFER(rchild))*
        increment length2
    endif
passivate
end
when receive y2
    decrement packets.waiting
    decrement length.2
    y := parent.packet(y2)
    found(y) := found(y2)
    if not root then
        send y to CHANNEL(SUPERVISOR(parent))*
    else
        destroy y
        notify output
    endif
passivate
end
```

3.3.2.2.4 Divide and Conquer

```
Inputs:  x, y1, y2
Outputs: x1, x2, xc, y
```

```
when receive input x
    increment packets.waiting
    x1 := lower(x)
    packet.id(x1) := 1
    x2 := upper(x)
    packet.id(x2) := 2
    send x1 to CHANNEL(BUFFER(lchild))*
    send x2 to CHANNEL(BUFFER(rchild))*
    increment length.1, length.2
    passivate
end
when receive yi, i=1, 2
    y := yi
    decrement length.i
    if packet.id(y) = 1 then
        left.packet(parent.packet(y)) = y
    else
        if packet.id(y) = 2 then
            right.packet(parent.packet(y)) = y
        endif
    endif
    if full(parent.packet(y)) then
        xc := parent.packet(y)
        send xc to the CHANNEL(BUFFER(compiler))*
    endif
passivate
```

```
end
when receive yc
    decrement packets.waiting
    if not root then
        send yc to CHANNEL(SUPERVISOR(parent))*
    else
        destroy yc
        notify output
    endif
passivate
end
```

3.3.2.3 Buffer Module

```
External Inputs:
        x
        buff.flushed
External Outputs:
        set.flushed.id
        hire.changing*
        fire.changing*
        normal.changing*
Internal Inputs:
        done
        buff.changing(work-type)
Internal Outputs:
        x
```

```
Initialize:
        queue := emtpy
        fcomp := free
        changing := false
when receive change(new-type) from EXEC
work-type := new-type
if work-type="inactive"
    if empty(que) and fcomp.free then
        send flushed to CHANNEL(BUFFER(parent))*
        self release from the system
    else
        changing := true
        passivate
    endif
else
    if work-type="f-computer"
        if parent exists then
            send changing(fire, id.side) to
                                    CHANNEL(EXEC(parent))*
        endif
        if full(register) then
            changing := false
            register.clear
            if not empty(que) then
                x := first(que)
                send x to FCOM
```

```
                que := rest(que)
                fcomp.free := false
            endif
        else
            changing := true
        endif
    else
        if work-type="coordinator"
            if parent exist then
               send changing(hire, id.side) to
                                    CHANNEL(EXEC(parent))*
            endif
            if not empty(que) or not fcomp.free
                changing := true
            endif
        endif
    endif
endif
passivate
end
when receive flushed.id (i=1, 2, c)
if changing and work-type = "f-computer" then
    register[i] := true
    reset child(i)
    if i=1 and coordinator.type="pipeline" then
        send fire.command to CHANNEL(EXEC(rchild))*
```

```
    else
        if coordinator.type="divide.and.conquer"
            if register[1] and register[2] and not
                                        register[3] then
               send fire.command to CHANNEL(EXEC(compiler))*
            endif
        endif
    endif
    if full(register) then
        register.clear
        changing := false
        send changing(normal, id.side) to
                                CHANNEL(EXEC(parent))*
        if not empty(que)
            x := first(que)
            send x to FCOM
            que := rest(que)
            fcomp.free := false
        else
            passivate
        endif
    endif
endif
passivate
end
```

```
when receive done
fcomp.free := true
if not changing then
    if not empty(que) then
        x := first(que) then
        send x to FCOM
        que := rest(que)
        fcomp.free:=false
    endif
else
    if work-type="inactive"
        if not empty(queue) then
            x := fisrt(que)
            send x to FCOM
            que := rest(que)
            fcomp.free:=false
            passivate
        else
            if not root
                send flushed to CHANNEL(BUFFER(parent))*
                self release from the system
            endif
        endif
    else
        if work-type="coordinator" then
            changing := false
```

```
            until empty(que)
                x := first(que)
                send x to SUPERVISOR
                que := rest(que)
            loop
        endif
    endif
endif
passivate
end
when receive x
if changing then
    insert(x, que)
else
    if work-type="f-computer"
        if empty(que) and fcomp.free then
            send x to FCOM
            fcomp.free:=false
        else
            insert(x, que)
        endif
    else
        if work-type="coordinator"
            send x to SUPERVISOR
        endif
   endif
```

```
endif
passivate
end
```

2.3.2.4 F-computer

```
External Outputs:
        y
Internal Inputs:
        x
Internal Outputs:
        done
when receive x
    computation-time := time-complexity(x)
    hold(computation-time)
    y := f(x)
    if fn.type = "full" or "upperhalf" then
        i := 1
    else
        if fn.type = "lowerhalf" then
            i := 2
        else
            if fn.type = "compiler" then
                i := c
            endif
        endif
     endif
```

```
        yi = y
        send yi to CHANNEL(SUPERVISOR(parent))*
        send done to BUFFER
passivate
end
```

CHAPTER 4

SIMULATION SET UP

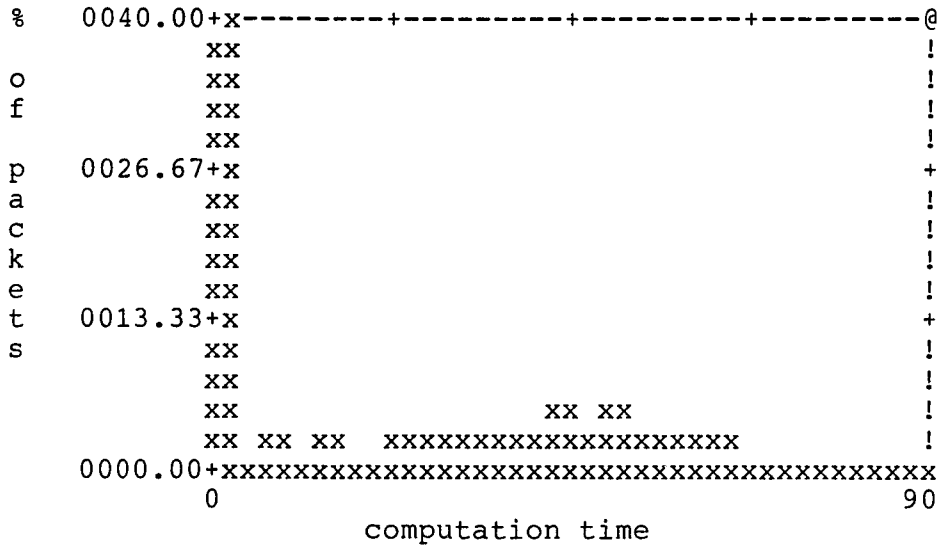
In this chapter, the simulation set up including the implementation, the packet characteristic, the packet arrival rate, the replications of experiment, the "hire-fire" signal generation, and the simulation environment are described.

4.1 Simulation Implementation

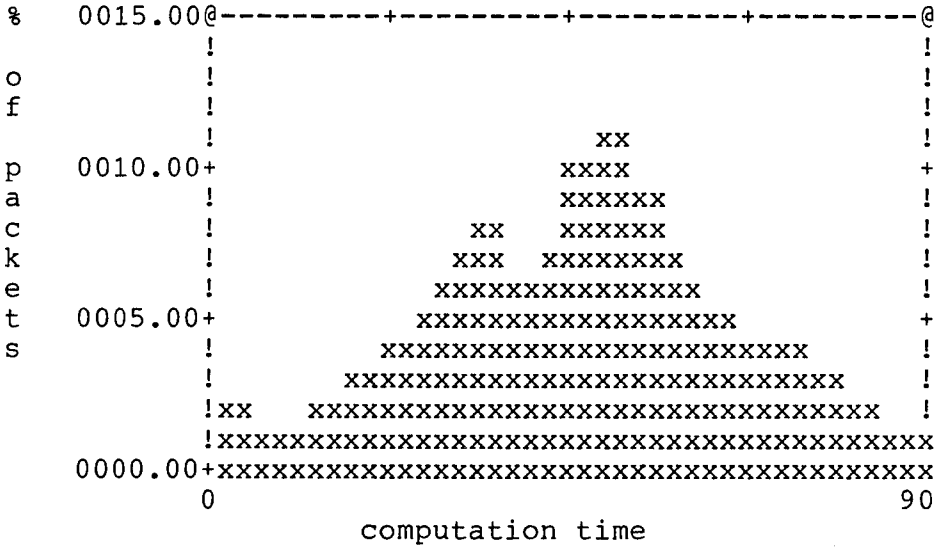
The simulation program is implemented in SIMSCRIPT II.5 programming language. Each fp, packet, or channel is a SIMSCRIPT entity which is a data structure with attributes to describe the current state of the entity.

4.2 Packet Characteristics

The packet in our experiments is described by the triple (l, u, j). The l, u, and j are generated by using the random number generator, "randi.f", in SIMSCRIPT II.5. The ranges of l, u, and j used in Linos' experiments [4] are [50, 100], [101, 150], and [1, 200] respectively. Kim [1] pointed out that the computation time of packets generated with these ranges is not evenly distributed (Figure 4.1 (i)). About 40% of the packets are of zero computation time. Kim suggested changing the ranges to [50, 100], [101, 150] and [1, 5000]



(i)



(ii)

Figure 4.1 Packet computation time distribution (i) Linos' case. (ii) Our case with new ranges and random number generators (7, 5, 8)

respectively. With these new values The distribution of the packet computation is much smoother and looks close to a normal distribution (Figure 4.2 (ii)).

4.3 Adaptive Policies

There are four types of policies -- the accept policy, the transmission policy, the selection policy, and the generation policy [4]. The policies used in the system do affect its overall behaviors and performance. The policies used in our model are summarized below.

4.3.1 Accept Policy

Each fp consults the accept policy to determine whether to accept the "hire" of the "fire signal or not. The "hire.limit" and the "fire.limit" which compare with the number of packets in the BUFFER's queue is used as criterion. For receiving a "hire" signal and the number of packets in the BUFFER's queue are more than the "hire.limit", the fp will carry out a "hire" operation. Similarly, a "fire" operation is performed if an "f-parent" receives a "fire" signal and the number of packets in the BUFFER's queue is less than the "fire.limit".

4.3.2 Transmission Policy

"Coordinators" consult transmission policies to determine how to propagate the "hire" and the "fire" signal properly to the next lower level. The "hire" signal is sent

to the processor with the longest BUFFER's queue, and, conversely, the "fire" signal is sent to the processor with the shortest queue.

4.3.3 Selection Policy

The system refers a "selection" policy to select the coordinator type in a "hire" operation. In our experiments, four different coordinator configurations are used: "multiserver", "pipeline", "divide'n'conquer", and "combination". In the first three configurations, only one related coordinator type is used throughout the simulation period. In the "combination" case, mixed types of coordinators are used and the type of coordinator is determined by a SIMSCRIPT random number generator "randi.f". Linos used all these three type coordinators in the "combination" -- "multiserver", "pipeline", and "divide'n'conquer" [4]. However, our experiment results show that the "pipeline" has the poorest performance and the "combination" performs poorly next to the "pipeline" when communication delay is considered. In this case, we believe that the performance of the "combination" is affected greatly by the anticipation of the "pipeline". Therefore, we remove the "pipeline" from the "combination" and the new "combination" now only consists of the "multiserver" and the "divide'n'conquer"; its performance did improve.

4.3.4 Generation Policy

The "hire" and the "fire" signals are generated in different types of fp's according the "generation" policy. The abilities of an fp generating these signals may be gained or lost depending on its type.

In our work, we can classify them into "centralized" and "distributed" generation policies. The "hire" signal is generated only "centrally" by the "root" of the system tree or "distributively" by the "f-computer" type fp's and the "fire" signal is generated only "centrally" by the "root" or "distributively" by the "f-parent" type fp's. There are four types policies used in our model: "root-root", "f.computer-root", "root-f.parent", and "f.computer-f.parent". The first fp type is responsible to generate "hire" signals and the second fp type is responsible to generate "fire" signals.

4.4 Replications of Experiments

Law (1986) pointed out that it is dangerous to make comparisons based on only one replication simulation for each system [2]. He also recommanded that, regardless of the cost per replication, at least three replications of the simulation are always to be made. In our experiments, five replications are made of each experiment.

These five replications are run by using five sets of different combinations of generators which produce the packets. There are ten random number generators in SIMSCRIPT

Table 4.1 The five random number generator triples and their average computation time

Set	Random Number Generator L	U	J	Avg Comp Time (units)
1	7	5	8	48.0880
2	7	9	8	48.8440
3	2	6	1	49.6540
4	9	1	7	50.3540
5	5	1	4	51.1100

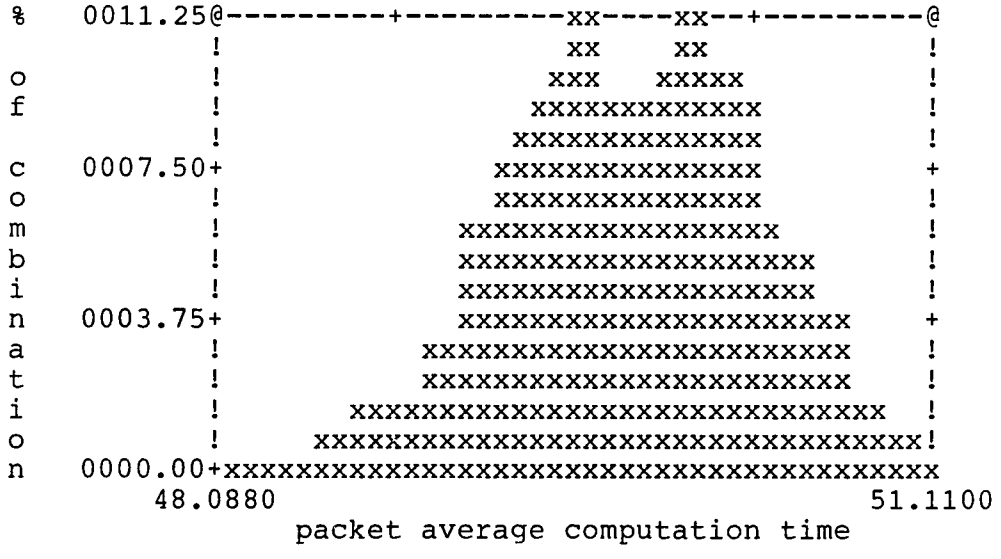


Figure 4.2 Distribution of the average packet computation time of the packet generation triples

II.5 [5] and there are 720 combinations of random number generators by taking 3 out of 10. The computation time distribution of these combinations is very close to a normal distribution (Figure 4.2). We choose triples, which represents the random number generators generating packet's lower bound, upper bound, and j value respectively, evenly spaced from the lowest computation time triple (7, 5, 8) to the highest computation time triple (5, 1, 4) (Table 4.1).

4.5 Packet Arrival Rate

In our experiments, the interarrival time of packets is a system parameter. We can choose either the fast or the slow arrival rate of the packets. The average computation time is about 50 time units in our experiments, and the interarrival time 2 time units and 9 time units are assigned to the fast and the slow rate respectively. Thus the interarrival time of the fast rate is about 1/25 of average computation and 1/6 for the slow rate.

4.6 Priority of Events and Processes

In the simulation programs, we use the SIMSCRIPT "event" in most cases just like Linos did. However the "process" is used in the CHANNEL module. We encountered a problem with tie breaking rules when events have scheduled to be activated at the same simulation time. In SIMSCRIPT II.5, the default priority of these "events" and "processes" activating at the same time is based on the order they were

scheduled. However, this priority order setting did not give exactly the same results of our model by setting zero communication delay with comparison to Linos' model. In order to verify the correctness of our model, a PRIORITY statement is put in Linos' program's preamble as well as ours, which assigns a priority to the "events" and the "processes", and yields exactly the same results as Linos' model. The priority order of these "events" and "processes" from the highest to the lowest is:

. generator (event): starting the simulation
. generate.tree.packets (event): generating packets
. repeator (process): scheduling "in.channel" process when a packet is submitted to the CHANNEL module
. in.channel (process): deferring a message in the CHANNEL
. hire.test (event): activating a "hire" signal
. fire.test (event): activating a "fire" signal
. generate.hold.time (event): processing a packet
. gen.compiler.time (event): compiling a packet
. print.performance (event): printing the current performance of the system.

In this priority scheme, first, the "generator" initiates the simulation and, second, the "generate.tree.packet" generates packets, which represents the environment. Third, messages transmitted in the CHANNELs are being scheduled their "in.channel" processes and, fourth, the "in.channel" processes elapsed with delay time

are activated and invoked at the destination fp's. Fifth, the "hire.test" and, then, "fire.test" are performed to generate the "hire" and the "fire" signals. Sixth, the packets being processed by "f-computer's" FCOM (generate.hold.time) and "compiler's" FCOM (gen.compiler.time) are activated. Finally, the "print.performance" outputs the system performance. Except for the "generator", which initiates the simulation, and the "generate.tree.packet", which generates packets for the system, the "repeator" and the "in.channel" have the higher priority over the others. That is, the messages going in or being in the "channel" will be activated first.

4.7 Hire-Fire Signal Generation Scheme

For each fp, only one "hire" or "fire" signal can be generated every test unit and there are two attributes serving the function of checking the scheduling of the "hire" and the "fire" test. The attributes, "hire.sched" and "fire.sched", are "event" pointers which are used to "cancel" the "hire" and the "fire" test when an fp is removed from the system. The "cancel" is a SIMSCRIPT statement which removes a scheduled "event" from the SIMSCRIPT "event" set [5]. If no "hire" or "fire" test is scheduled in this fp, the attribute ".sched" is set to zero which means pointing to nothing. Before a "hire" or a "fire" test is scheduled, the fp tests the following attributes:

. ".sched" pointer

. "m.ev.s" of the ".sched" if ".sched" is not zero

The schedule is done if the ".sched" is zero which indicates that no "event" is scheduled yet. Otherwise, the "m.ev.s" attribute of the "event" pointed by ".sched" is tested. If "m.ev.s" is zero, which means the "event" is not in the event list, a new test "event" is scheduled. The "m.ev.s", an automatic "event" attribute of SIMSCRIPT, indicates whether the "event" is in the event list or not.

4.8 Simulation Facilities

In our experiments, we use VAX 11/780 supermini computer to run our simulation program. We also tried to run the program of IBM personal computer and we got the same results as running on VAX. However, PC seems to be too slow to run such a complicated simulation program. For example, it may take several hours (we did not even have patience to finish) to run an experiment of 500 packets of the "divide'n'conquer" coordinator configuration with comparison of four or five minutes on the VAX.

4.9 Simulation Set Up Summary

The settings of our experiments are summarized as follows:

. number of packets: 500 (in most cases) and 3000 (in the experiments of batch average turn around time)

. hire limit: 5 (packets)*

. fire limit: 3 (packets)*

* these criteria are used to compare with the number of packets in the BUFFER's queue.

. packet triple ranges:

l : 50 -- 100

u : 101 -- 150

j : 1 -- 5000

. packet generator triples: (7, 5, 8), (7, 9, 8), (2, 6, 1), (9, 1, 7), (5, 1, 4)

. packet arrival rates: fast (packets arrive every 2 units), slow (packets arrive every 9 units), fast-to-slow

. hire/fire test scheduling interval: 10 units

. types of coordinators: "multiserver", "pipeline", "divide'n'conquer", and "combination" (mixture of "multiserver" and "divide'n'conquer" only, refer to section 4.3.3)

. hire/fire generation policies: "root-root", "root-Fp", "Fc-root", "Fc-Fp"

. communication delay: 0, 2.5, 5.0, 7.5, 10.0 units

CHAPTER 5

SIMULATION RESULTS AND ANALYSIS

5.1 Experiment Objectives

The objectives of our experiments are to study:

- Number of processors used in the system against time
- Throughput
- Average packet turn around time
- Batch average turn around time

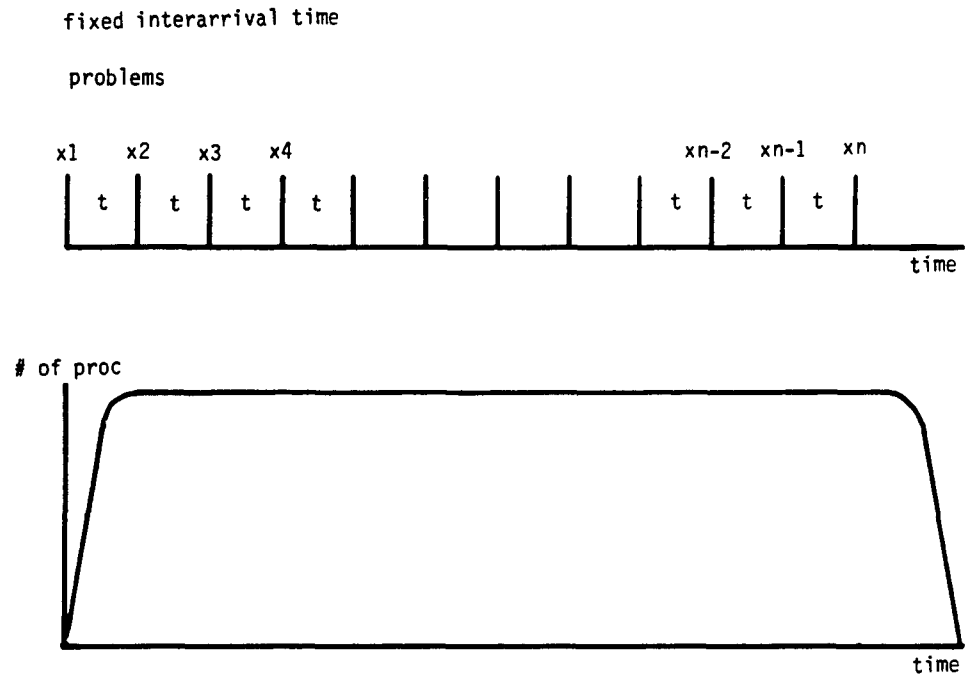
We performed the experiments in different environments -- three packet arrival rates and five communication delays (we also use "CD" for short), as well as with different system configurations -- four coordinator types and four "hire/fire" generation policies. Moreover, for each system configuration, the simulations were repeated five times with different sets of random number generators which generate the packets. The results shown later are the average of these five replications. With these results, we can see how the system behaves with different configurations and environments.

5.2 Number of Processors Used

The number of processors used in the system changing over time reflects the performance of the "hire/fire" policies.

In our experiments, the packets arrives at a fixed rate and whose computation time is uniformly or normally distributed in the interval [0, 100]. Thus, in this case, an ideal curve (Figure 5.1 (i)) of number of processors plotted against time is that it rises fast, stays in a stable state, and, after all the problems are done, drops fast too. Rising fast means that the system is able to adjust its own configuration to accomodate to the changing environment by "hiring" more "helpers". Staying in a stable state indicates that the system is stable and is able to handle the environment well without "hiring" or "firing" any "helpers". The stable state is called steady state. After the system processes all the problems, it does not need the "helpers" anymore and should be able to "fire" them quickly.

In our experiments, we found that the number of processors may not follow the ideal curve shown above. The same system configuration and same packet arrival rate may yield different behaviors with different packet random number generators. We concluded that there are ten different behaviors of the number of processors against time (Figure 5.1). Note that the diagrams in Figure 5.1 are not of the same scale. In Figure 5.1 and later, the "Fc" stands for "f-computer" and the "Fp" for "f-parent." The "policy" means "hire-fire" signal generation policy; the first one generates "hire" and the second one generates "fire". For



(i)

Figure 5.1 (i) An ideal curve of the number of processors used in our experiments. (ii) "root-root" policy. (iii) "root-Fp" policy. (iv) "Fc-root" policy. (v) "Fc-Fp" policy. (vi) "pipeline" using "root-root" policy. (vii) "pipeline" using "Fc-root" policy (to be continued)

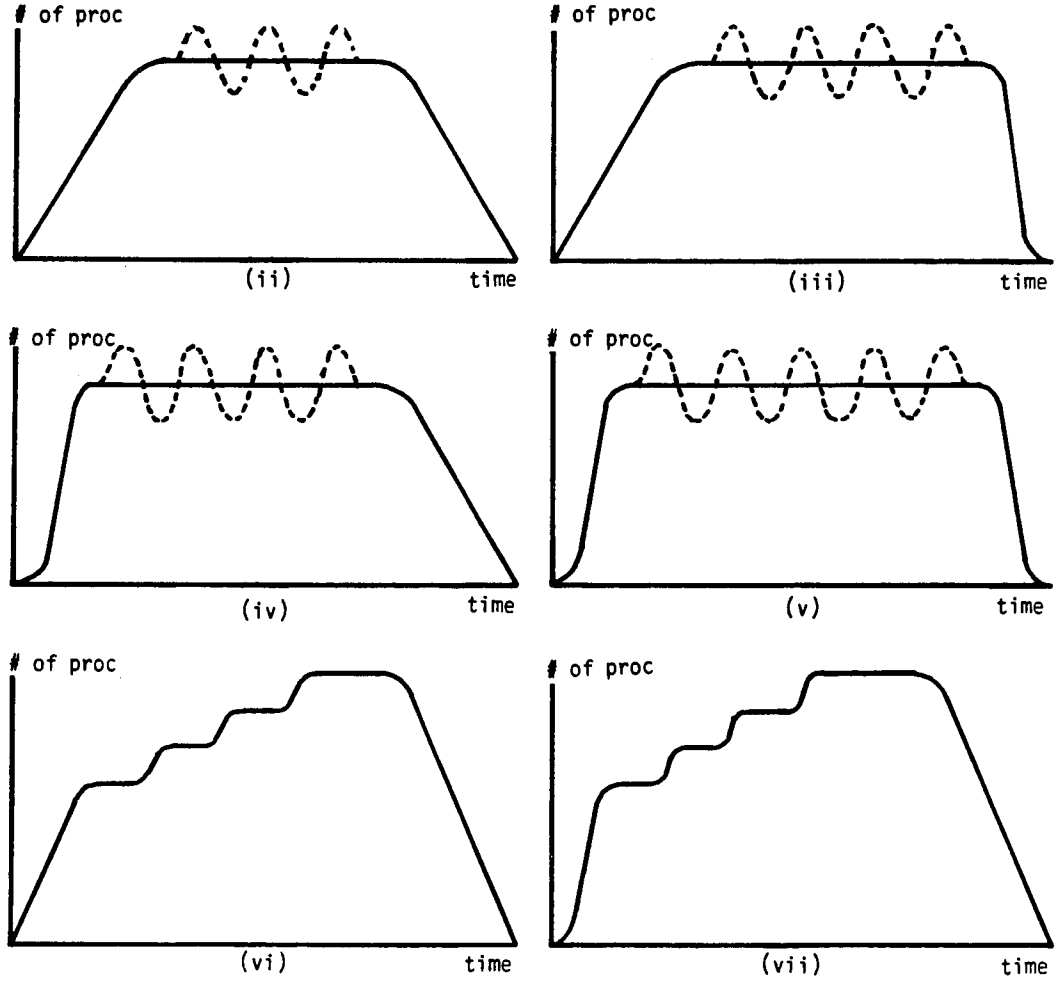


Figure 5.1 (Continued)

example, the "root-Fp" policy indicates that "root" generates "hire" signal and the "f-parent" type fp's are responsible to produce "fire" signals.

In the case of "centralized" generation policies, in which the "root" is responsible to generate the "hire" or the "fire" signals, only one "hire" or one "fire" signals are generated periodically at the root and propagated downward to the system tree. Therefore, at most one "hire" or "fire" operation can be performed at the lower level. In other words, the number of processors is only a constant increment or decrement every "hire/fire" generation units; the rising and the falling edges are very close to a straight line (Figure 5.1 (ii), (iii), and (iv)).

In the "distributed" generation policies, in which the "f-computers" are responsible to generate "hire" signals, and the "f-parents" produce "fire" signals, more than one "hire" or "fire" signals can be generated every "hire/fire" generation units and more than one "hire" or "fire" operations may be carried out at the same time. At the beginning of simulation, the system needs to "hire" a lot of fp's because of the increasing incoming work load. Several "hire" operations can be done at the same time and the number of processors increases very fast. Later on, the system has almost enough fp's and, therefore, only few "hire" operations are performed. The number of processor increases slowly as it is entering "saturation" (Figure 5.1

(iv) and (v)). Similarly, after the system has processed all the problems, several "fire" operations are done at the same time, the number of processor drops quickly. However, the shorter the system tree is, the fewer "fire" operations are performed because the number of "f-parents" decreases; hence the curve flattens out (Figure 5.1 (iii) and (v)).

Table 5.1 shows the related curves for different system configurations and environments. Note that in some cases, the number of processors oscillates instead of being stable at steady state. This is especially likely to occurs under the system configuration with the distributed "fire" generation policy. In some cases, the oscillation takes place only for some specific communication delays. For the case 7, 15, 18, 19, 20, 26, 27, and 28, the oscillations occur only for zero communication delay. In the cases 23 and 31, the number of processors stops oscillating when the communication delay is 10 time units. In case 2 and 10, the steplike curves indicate that the "pipeline" coordinator systems seems to be repeatedly moving up from one stable state to next one until all the problems are done (Figure 5.1 (vi) and (vii)). The "hire/fire" generation policies always affect the shapes of the curves.

Table 5.1 The distributions of the number of processors with different system configurations and environments

Case	Hire	Fire	Coord	Arrival	Fig.5.1	Max # of Proc
01	Root	Root	Ms	Fast	(ii)	62.44
02			Pl	Fast	(vi)	41.00
03			Dq	Fast	(ii)	63.40
04			Comb	Fast	(ii)	79.56
05	Root	Fp	Ms	Fast	(iii)*	62.44
06			Pl	Fast	(iii)*	37.48
07			Dq	Fast	(iii)**	63.40
08			Comb	Fast	(iii)*	75.14
09	Fcom	Root	Ms	Fast	(iv)	62.60
10			Pl	Fast	(vii)	40.60
11			Dq	Fast	(iv)	63.40
12			Comb	Fast	(iv)	109.16
13	Fcom	Fp	Ms	Fast	(v)*	62.60
14			Pl	Fast	(v)*	37.72
15			Dq	Fast	(v)**	63.40
16			Comb	Fast	(v)*	99.92
17	Root	Root	Ms	Slow	(ii)	14.44
18			Pl	Slow	(ii)**	27.56
19			Dq	Slow	(ii)**	49.96
20			Comb	Slow	(ii)**	34.36
21	Root	Fp	Ms	Slow	(iii)*	15.00
22			Pl	Slow	(iii)*	21.28
23			Dq	Slow	(iii)***	50.32
24			Comb	Slow	(iii)*	38.00
25	Fcom	Root	Ms	Slow	(iv)	14.04
26			Pl	Slow	(iv)**	27.96
27			Dq	Slow	(iv)**	48.76
28			Comb	Slow	(iv)**	35.08
29	Fcom	Fp	Ms	Slow	(v)*	15.00
30			Pl	Slow	(v)*	21.08
31			Dq	Slow	(v)***	49.96
32			Comb	Slow	(v)*	37.00

where:

* the number of processors oscillates instead of being steady state (oscillation is shown as dashed line).

** the number of processors oscillates only at zero communication delay.

*** the number of processor oscillates except the case communication delay is 10.

5.3 Throughput

Throughput is defined as the number of solved problems divided by the time period needed to solve the problems:

$$\text{throughput} = \frac{\text{number of solved problems}}{\text{time period to solve the problems}}$$

Throughput is a measurement of system performance. The higher it is, the better is the performance. Our experimental results are shown in Table 5.2 and 5.3. In general, the higher the communication delay, the lower the throughput will be. Figure 5.2 to 5.9 illustrate how the communication delay affects the behavior of the system throughput. In most cases of fast arrival rate, the throughput decreases almost linearly as the communication delay increases. For the slow arrival rate, the increasing communication delay does not affect the "multiserver", the "divide'n'conquer", and the "combination" too much; they almost remain the same even though the communication delay increases. On the other hand, the "pipeline" is more sensitive to the increasing communication delay; it drops much faster than the others. Note that in some cases, for example Figure 5.2 and 5.5, the "multiserver's" throughput first increases before decreasing with communication delay, which we did not understand well.

Table 5.2 Throughput of coordinators with (i) Root-Root, (ii) Root-Fp, (iii) Fc-Root, and (iv) Fc-Fp generation policy at fast arrival rate

Com Del	Ms	Pl	Dq	Comb
00.0	0.409	0.312	0.384	0.415
02.5	0.418	0.272	0.373	0.401
05.0	0.420	0.256	0.362	0.388
07.5	0.406	0.230	0.352	0.377
10.0	0.391	0.217	0.342	0.364

(i)

Com Del	Ms	Pl	Dq	Comb
00.0	0.403	0.288	0.381	0.403
02.5	0.387	0.259	0.373	0.393
05.0	0.372	0.241	0.362	0.385
07.5	0.377	0.220	0.352	0.372
10.0	0.367	0.200	0.342	0.363

(ii)

Com Del	Ms	Pl	Dq	Comb
00.0	0.461	0.312	0.384	0.470
02.5	0.443	0.270	0.373	0.465
05.0	0.431	0.254	0.362	0.448
07.5	0.424	0.226	0.352	0.420
10.0	0.414	0.214	0.342	0.400

(iii)

Com Del	Ms	Pl	Dq	Comb
00.0	0.381	0.288	0.381	0.438
02.5	0.397	0.263	0.373	0.437
05.0	0.387	0.242	0.362	0.427
07.5	0.366	0.219	0.352	0.410
10.0	0.374	0.202	0.342	0.401

(iv)

Table 5.3 Throughput of coordinators with (i) Root-Root, (ii) Root-Fp, (iii) Fc-Root, and (iv) Fc-Fp generation policy at slow arrival rate

Com Del	Ms	Pl	Dq	Comb
00.0	0.108	0.109	0.109	0.109
02.5	0.109	0.106	0.110	0.109
05.0	0.108	0.104	0.109	0.109
07.5	0.108	0.101	0.108	0.108
10.0	0.108	0.099	0.106	0.107

(i)

Com Del	Ms	Pl	Dq	Comb
00.0	0.106	0.107	0.109	0.106
02.5	0.106	0.106	0.109	0.107
05.0	0.105	0.105	0.108	0.107
07.5	0.105	0.104	0.108	0.106
10.0	0.105	0.101	0.106	0.106

(ii)

Com Del	Ms	Pl	Dq	Comb
00.0	0.108	0.109	0.109	0.108
02.5	0.108	0.106	0.110	0.109
05.0	0.108	0.103	0.109	0.109
07.5	0.107	0.101	0.108	0.108
10.0	0.108	0.099	0.106	0.107

(iii)

Com Del	Ms	Pl	Dq	Comb
00.0	0.106	0.107	0.109	0.107
02.5	0.105	0.106	0.109	0.108
05.0	0.106	0.105	0.108	0.108
07.5	0.106	0.104	0.108	0.107
10.0	0.105	0.101	0.106	0.106

(iv)

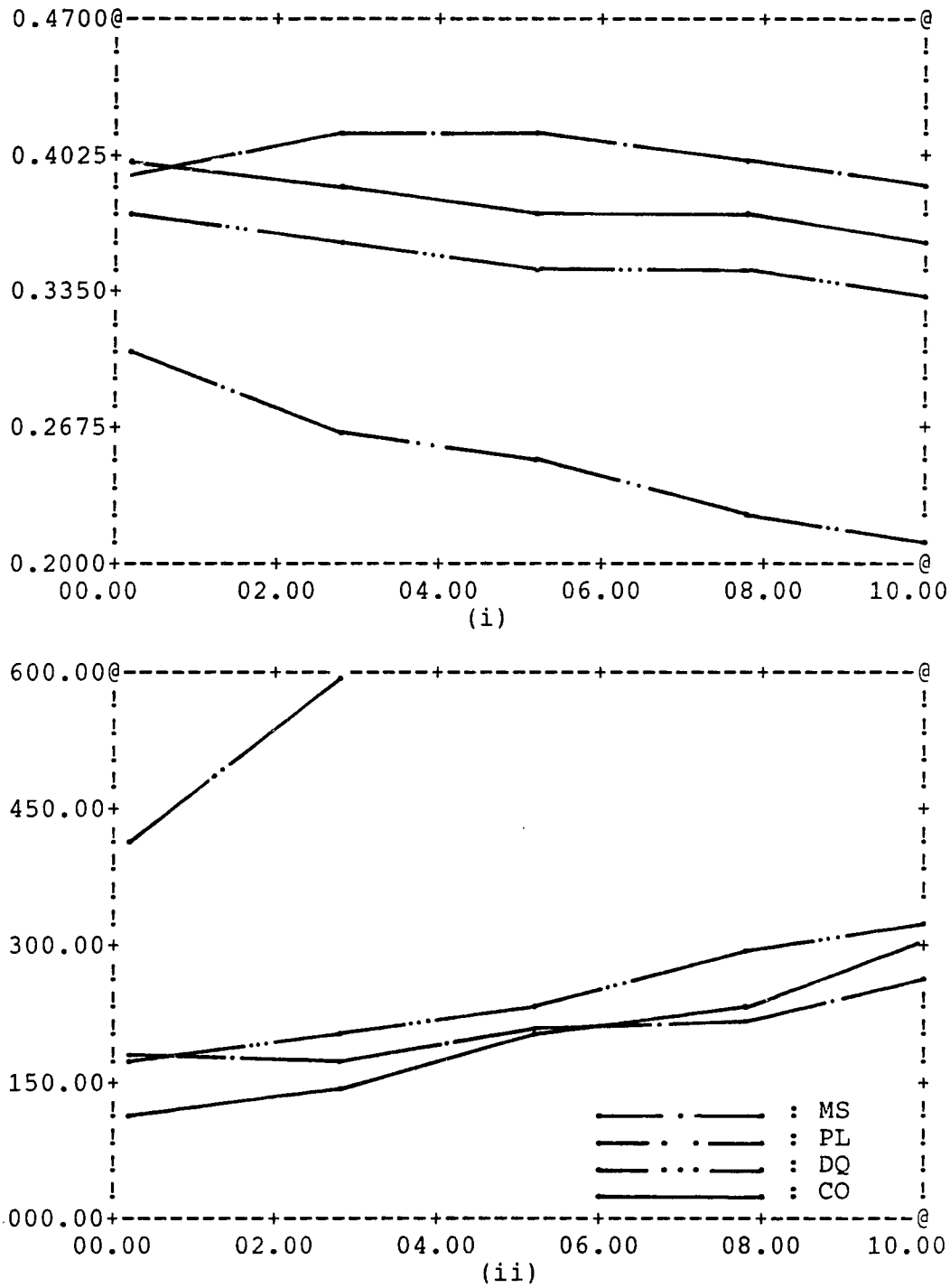


Figure 5.2 (i) Throughput vs CD and (ii) average TA vs CD with "root-root" policy at fast arrival rate

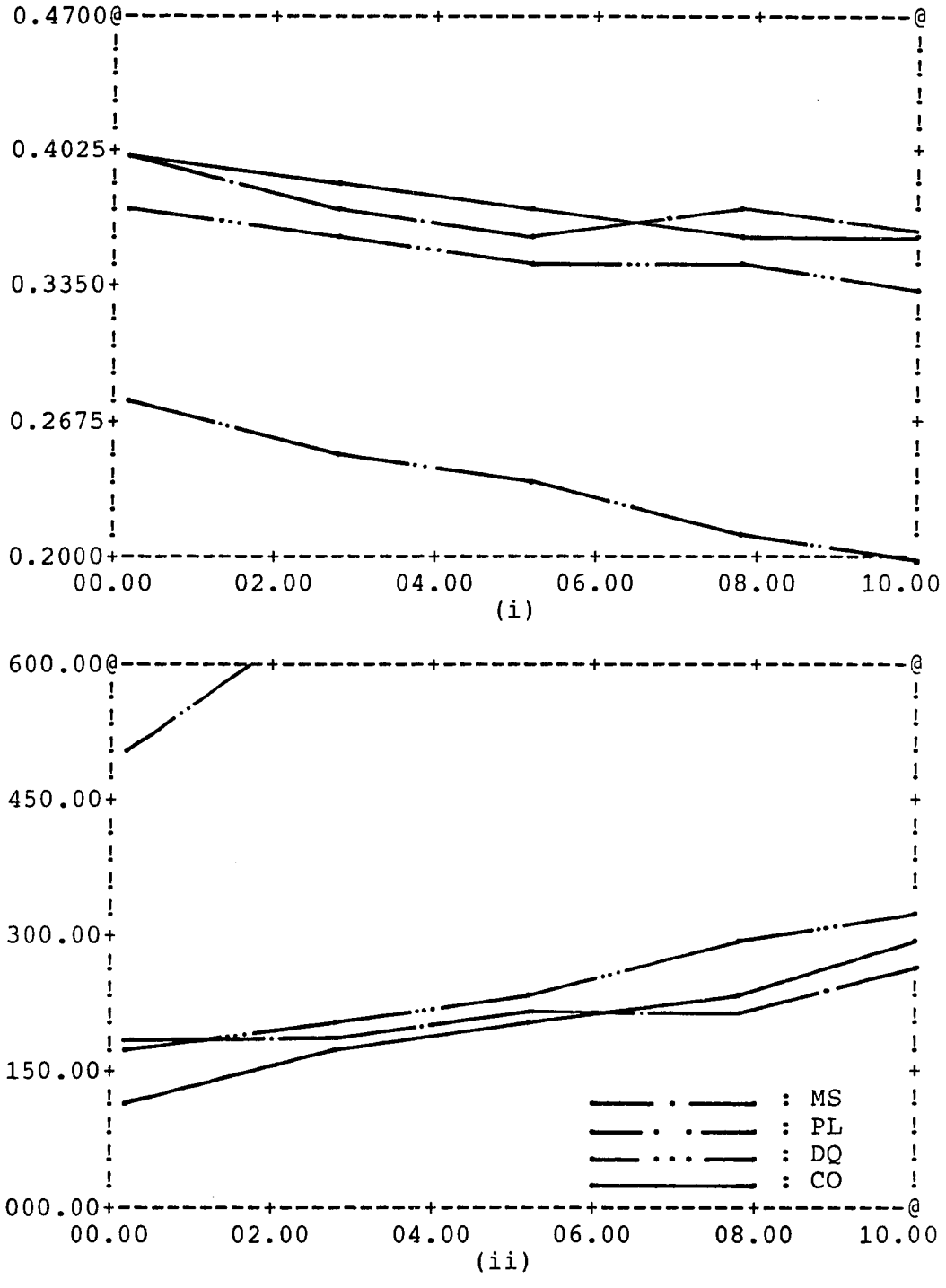


Figure 5.3 (i) Throughput vs CD and (ii) average TA vs CD with "root-Fp" policy at fast arrival rate

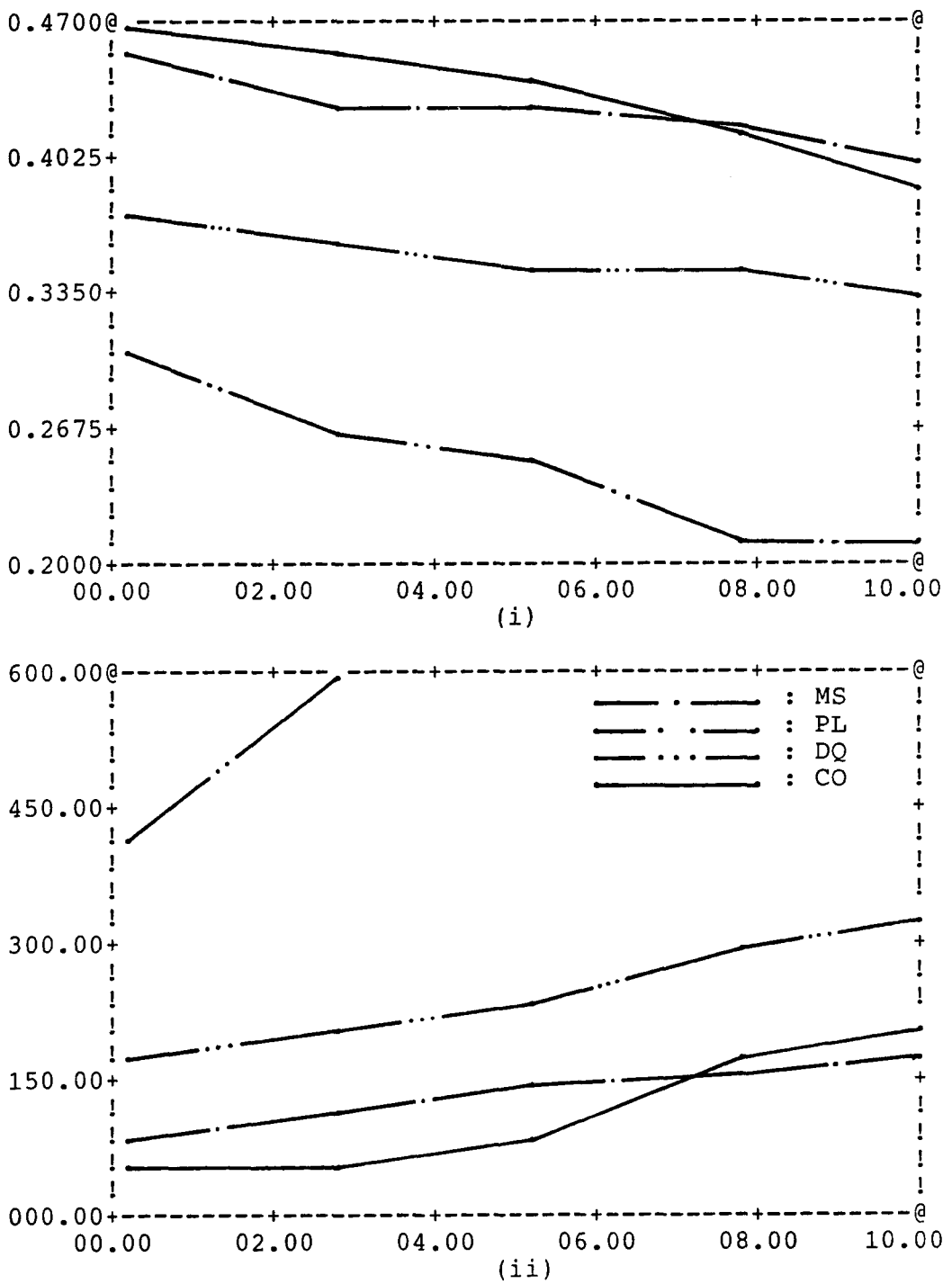


Figure 5.4 (i) Throughput vs CD and (ii) average TA vs CD with "Fc-root" policy at fast arrival rate

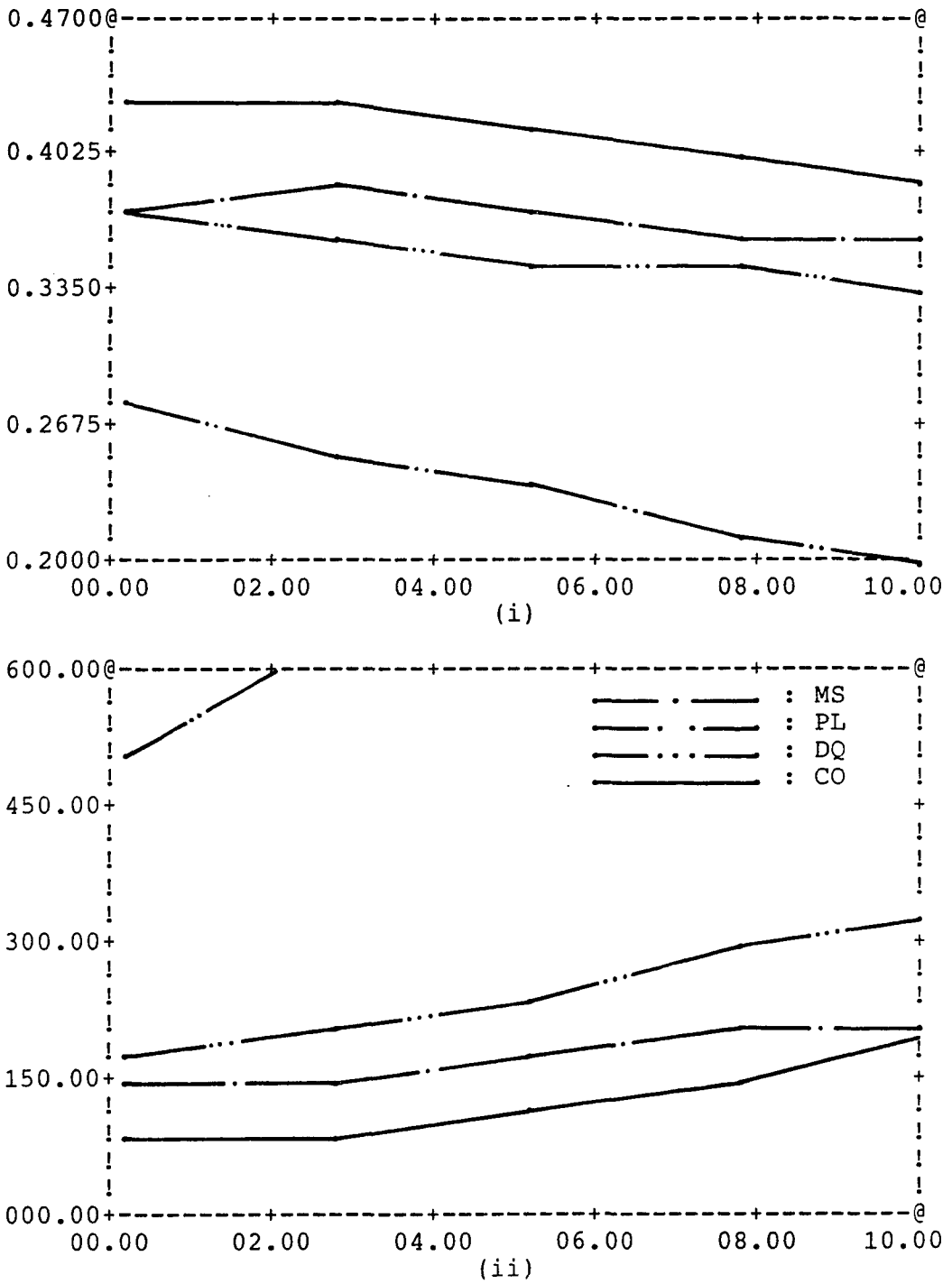


Figure 5.5 (i) Throughput vs CD and (ii) average TA vs CD with "Fc-Fp" policy at fast arrival rate

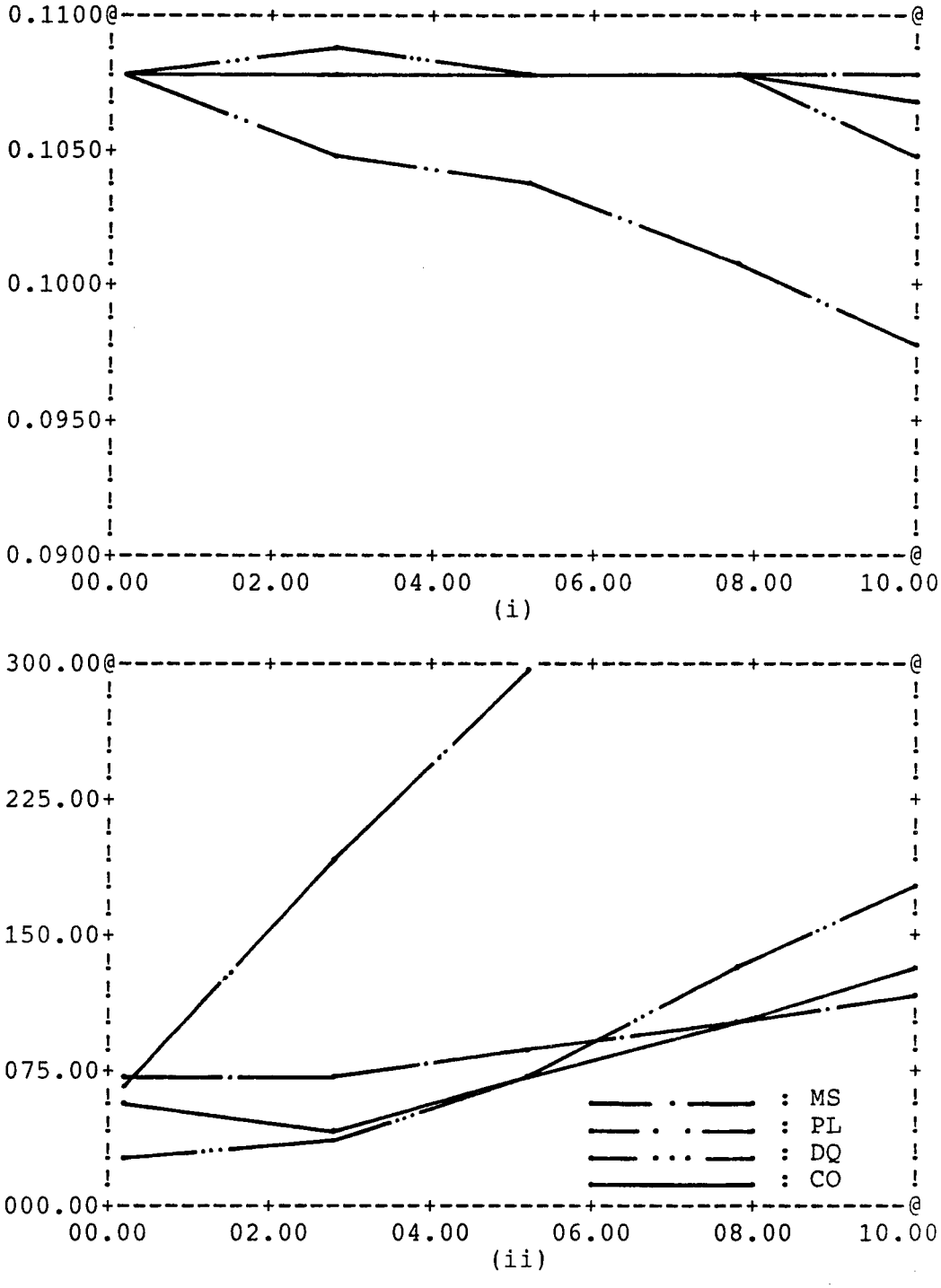
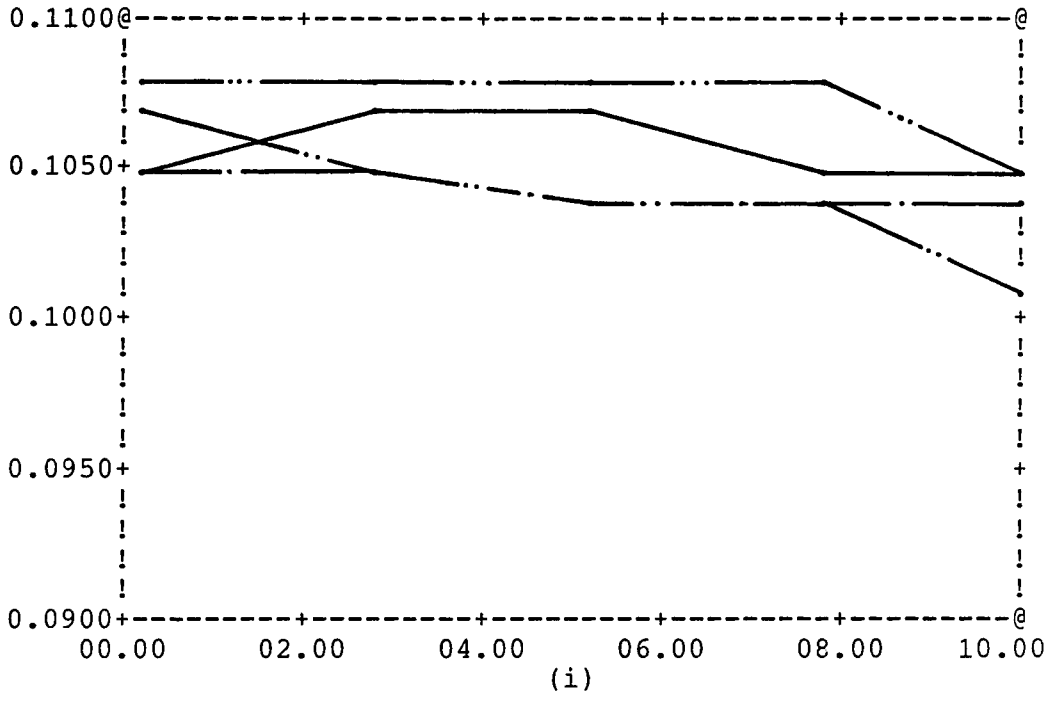


Figure 5.6 (i) Throughput vs CD and (ii) average TA vs CD with "root-root" policy at slow arrival rate



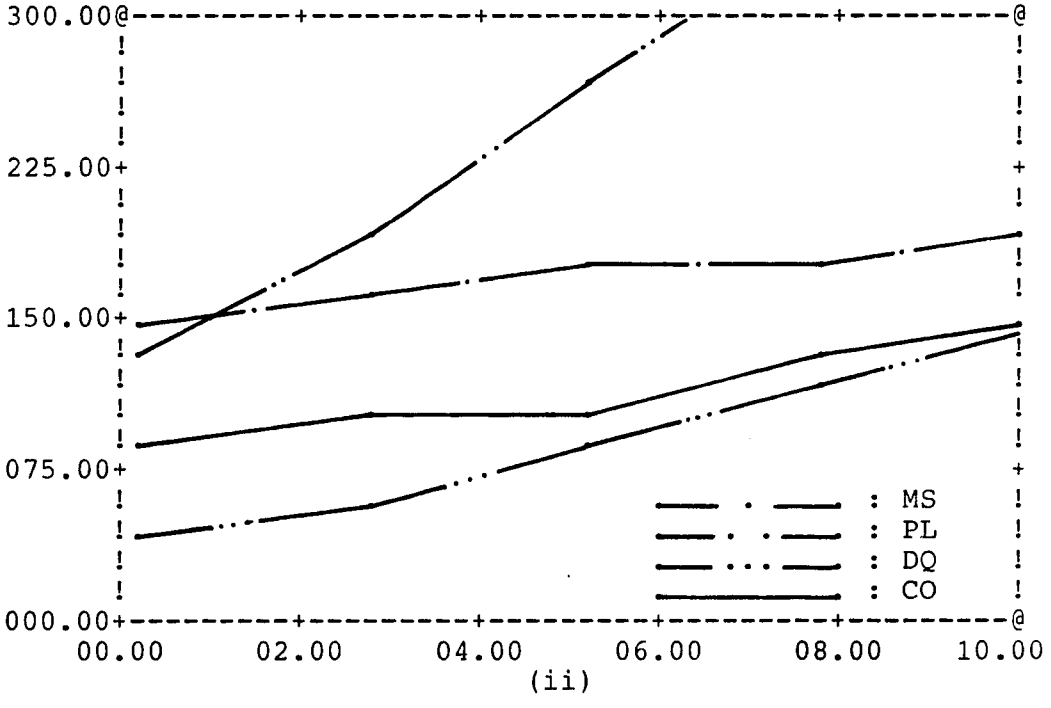


Figure 5.7 (i) Throughput vs CD and (ii) average TA vs CD with "root-Fp" policy at slow arrival rate

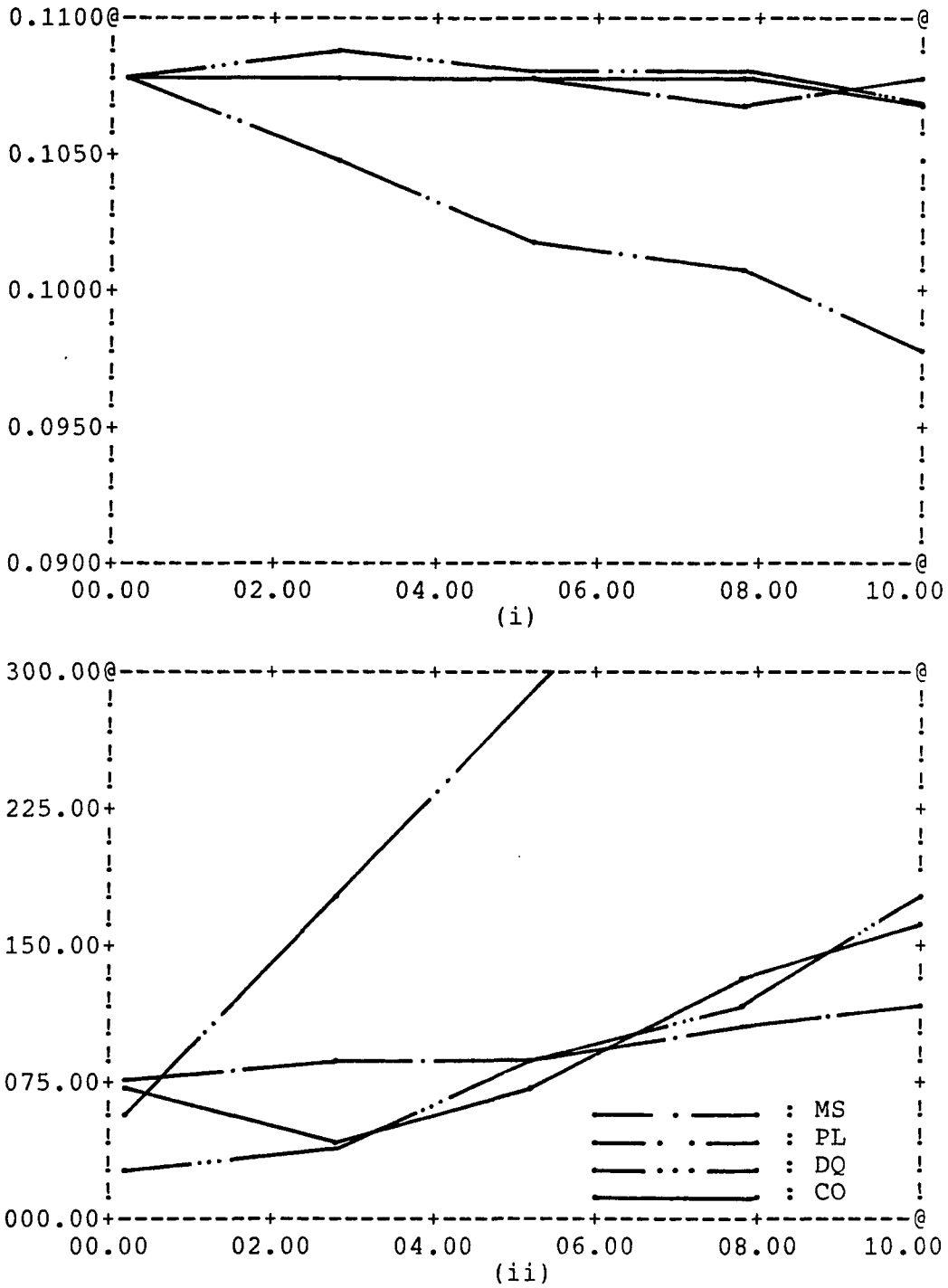


Figure 5.8 (i) Throughput vs CD and (ii) average TA vs CD with "Fc-root" policy at slow arrival rate

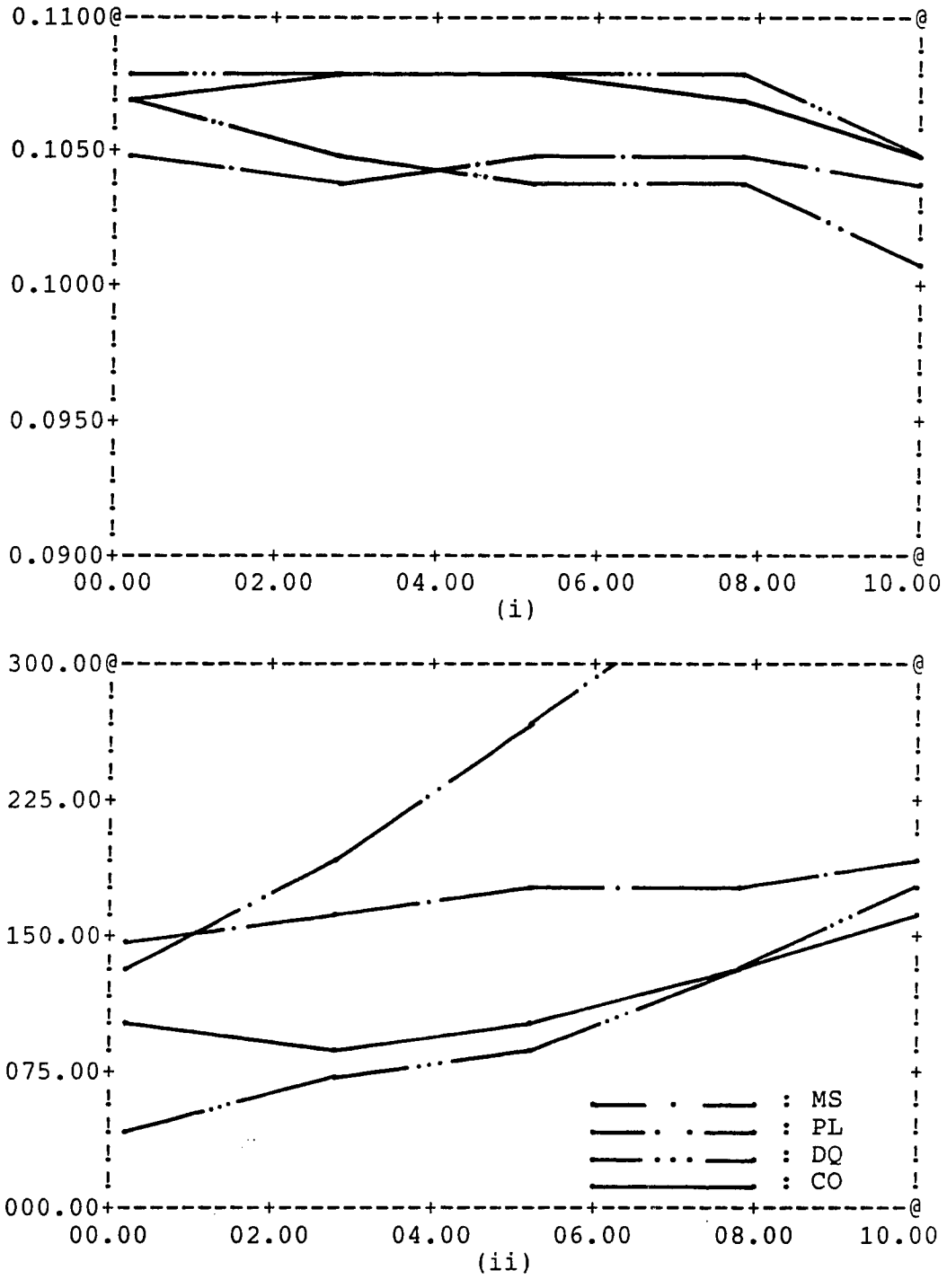


Figure 5.9 (i) Throughput vs CD and (ii) average TA vs CD with "Fc-Fp" policy at slow arrival rate

5.4 Average Packet Turn Around Time

The average packet turn around time is measured as follows:

- when the system receives a packet from the outside world, the packet arrival time (system clock at this moment) is recorded in an attribute of the packet.
- when the packet is ready to leave the system, the packet turn around time, the period of the packet staying in the system (the system clock minus the time attribute of the packet), is added to a variable which hold the sum of each packet's turn around time. The counter recording the number of packets processed is incremented by one.
- when all the packets are processed, the average packet turn around time is obtained by dividing the total packet turn around time by the number of processed packets.

The average packet turn around time (we will use "turn around time" for short) is also used to measure the performance of the system. The higher the turn around time, the poorer the system performs. Our experimental results (Table 5.4 to 5.5 and Figure 5.2 to 5.9) show that, in general, the turn around time increases as the communication delay increases. The increment is almost linear especially for the "pipeline" case.

For the fast arrival rate, the "pipeline" has the longest turn around time in all cases and the "combination" has the shortest turn around time in most cases. In the

Table 5.4 Turn around time of coordinators with (i) Root-Root, (ii) Root-Fc, (iii) Fc-Root, and (iv) Fc-Fp generation policy at fast arrival rate

Com Del	Ms	Pl	Dq	Comb
00.0	197.0	447.0	180.5	132.2
02.5	203.5	619.8	221.8	174.8
05.0	216.4	754.0	263.6	215.7
07.5	249.2	926.9	305.8	258.3
10.0	277.8	1072.3	348.5	301.5

(i)

Com Del	Ms	Pl	Dq	Comb
00.0	198.2	519.2	182.8	145.8
02.5	209.4	675.9	221.8	185.1
05.0	227.7	807.4	263.6	222.0
07.5	252.9	976.6	305.8	262.0
10.0	281.6	1188.9	348.5	302.8

(ii)

Com Del	Ms	Pl	Dq	Comb
00.0	118.5	447.0	180.5	75.8
02.5	134.5	628.5	220.9	68.4
05.0	157.0	757.0	262.7	110.2
07.5	180.9	937.5	305.0	184.4
10.0	206.6	1086.4	347.7	229.7

(iii)

Com Del	Ms	Pl	Dq	Comb
00.0	155.3	519.2	182.8	95.8
02.5	175.8	649.2	220.9	96.4
05.0	196.5	802.5	262.7	125.5
07.5	217.3	976.2	305.0	177.6
10.0	239.3	1162.5	347.7	226.6

(iv)

Table 5.5 Turn around time of coordinators with (i) Root-Root, (ii) Root-Fc, (iii) Fc-Root, and (iv)Fc-Fp generation policy at slow arrival rate

Com Del	Ms	Pl	Dq	Comb
00.0	88.9	70.4	42.8	63.6
02.5	89.0	196.7	50.0	57.9
05.0	99.2	313.9	79.4	80.5
07.5	114.2	424.3	143.3	113.8
10.0	128.8	550.7	181.7	135.7

(i)

Com Del	Ms	Pl	Dq	Comb
00.0	162.8	148.5	58.2	99.5
02.5	172.7	204.2	73.3	111.2
05.0	181.1	282.2	97.5	105.4
07.5	193.5	349.1	125.8	143.8
10.0	204.6	435.7	155.0	161.4

(ii)

Com Del	Ms	Pl	Dq	Comb
00.0	88.7	70.4	42.8	82.8
02.5	100.4	193.3	48.2	58.1
05.0	97.9	324.5	97.4	86.8
07.5	120.5	445.6	132.1	137.0
10.0	124.2	551.2	181.7	167.8

(iii)

Com Del	Ms	Pl	Dq	Comb
00.0	162.7	148.5	58.2	114.4
02.5	173.0	203.7	75.5	104.3
05.0	180.8	282.4	102.1	111.2
07.5	189.7	350.9	136.6	136.0
10.0	201.9	482.8	181.7	167.9

(iv)

cases of slow arrival rate, in most situations, the "divide'n'conquer" performs best. The "pipeline" has the highest slope in the curve of TA against delay while the "multiserver" has the lowest slope; the "pipeline" is most sensitive to communication delay, and the "multiserver" is most insensitive to it. The "pipeline" has the worst turn around time performance when communication delay is not zero. The "combination" has smaller slope than the "divide'n'conquer" does in the slow arrival rate cases, and they have almost the same slope in the fast arrival rate cases.

The comparisons of coordinator types are summarized in Table 5.6 and 5.7. When the communication delay is zero, the "divide'n'conquer" is the best and the "pipeline" is the worst in the slow arrival rate, but the "multiserver" is the worst and the "combination" is the best in the fast arrival rate. This observation is consistent with Liaw's findings [3]. Recall however, that in our experiment, the "combination" only consists of the "multiserver" and the "divide'n'conquer".

Throughout our experiments, the "pipeline" is proven the worst. For the fast arrival rate, the "combination" is the best with communication delay ranging from 0 to 5.0 and the "multiserver" is best in the cases of the "root-root", the "root-Fp", and the "Fc-root" generation policy with communication delay ranging from 7.5 to 10.0. For the slow

Table 5.6 Comparisons of coordinators' performance at fast arrival rate

Gen Plcy			# best comparison					# worst comparison				
			CASE					CASE				
Hire	Fire	Coord	1	2	3	4	5	1	2	3	4	5
Root	Root	MS		*	*	**	**					
		PL						oo	oo	oo	oo	oo
		DQ										
		CO	**	*	*							
Root	Fp	MS	*			**	**					
		PL						oo	oo	oo	oo	oo
		DQ										
		CO	**	**	**							
Fc	Root	MS				**	**					
		PL						oo	oo	oo	oo	oo
		DQ										
		CO	**	**	**							
Fc	Fp	MS										
		PL						oo	oo	oo	oo	oo
		DQ										
		CO	**	**	**	**	**					

* : # of the best performance from turn-around time and throughput
o : # of the worst performance from turn-around time and throughput
Case 1: communication delay = 0.0 units
Case 2: communication delay = 2.5 units
Case 3: communication delay = 5.0 units
Case 4: communication delay = 7.5 units
Case 5: communication delay = 10.0 units

Table 5.7 Comparisons of coordinators' performance at slow arrival rate

Gen Plcy			# best comparison CASE					# worst comparison CASE				
Hire	Fire	Coord	1	2	3	4	5	1	2	3	4	5
Root	Root	MS				*	**	oo				
		PL	*						oo	oo	oo	oo
		DQ	*	**	**	*						
		CO	**		*	**						
Root	Fp	MS						oo	o	o		
		PL							oo	oo	oo	oo
		DQ	**	**	**	**	**					
		CO					*	o				
Fc	Root	MS				*	**	oo				
		PL	*						oo	oo	oo	oo
		DQ	**	**	*	*						
		CO			**	*		o				
Fc	Fp	MS						oo	o			
		PL							o	oo	oo	oo
		DQ	**	**	**	*	*					
		CO			*		*	**				

* : # of the best performance from turn-around time and throughput
o : # of the worst performance from turn-around time and throughput
Case 1: communication delay = 0.0 units
Case 2: communication delay = 2.5 units
Case 3: communication delay = 5.0 units
Case 4: communication delay = 7.5 units
Case 5: communication delay = 10.0 units

arrival rate, the "divide'n'conquer" is the best in most cases. When the communication delay is 10, the "multiserver" is the best with the "root-root" and the "Fc-root" generation policy, and the "combination" is the best with the "Fc-Fp" generation policy.

The comparisons of different generation policies are also done, which are illustrated in Figure 5.10 to 5.17 and Table 5.8 and 5.9. In the fast arrival rate (Table 5.9), the best policy for the "multiserver", the "divide'n'conquer", and the "combination" is the "Fc-root" and the "root-Fp" is the worst. It is interested to point out that, in the "divide'n'conquer" case, no matter what types of policies are used, the measurements of the system's performance are very close to each other (Figure 5.12). In the case of the "pipeline", it works best with the "root-root" policy and worst with the "root-Fp".

In the cases of slow arrival rate (Table 5.9), the "root-root" policy works best and the "root-Fp" works poorest with the "multiserver" and the "combination". In the case of the "pipeline", the "root-root" is the best and the "root-Fp" is the worst. It is interested to see that the performances of the "root-root" and the "Fc-root" are very close, and so are the "root-Fp" and the "Fc-Fp" (Figure 5.15). The "divide'n'conquer" performs best with the "Fc-root" policy and worst with the "Fc-Fp".

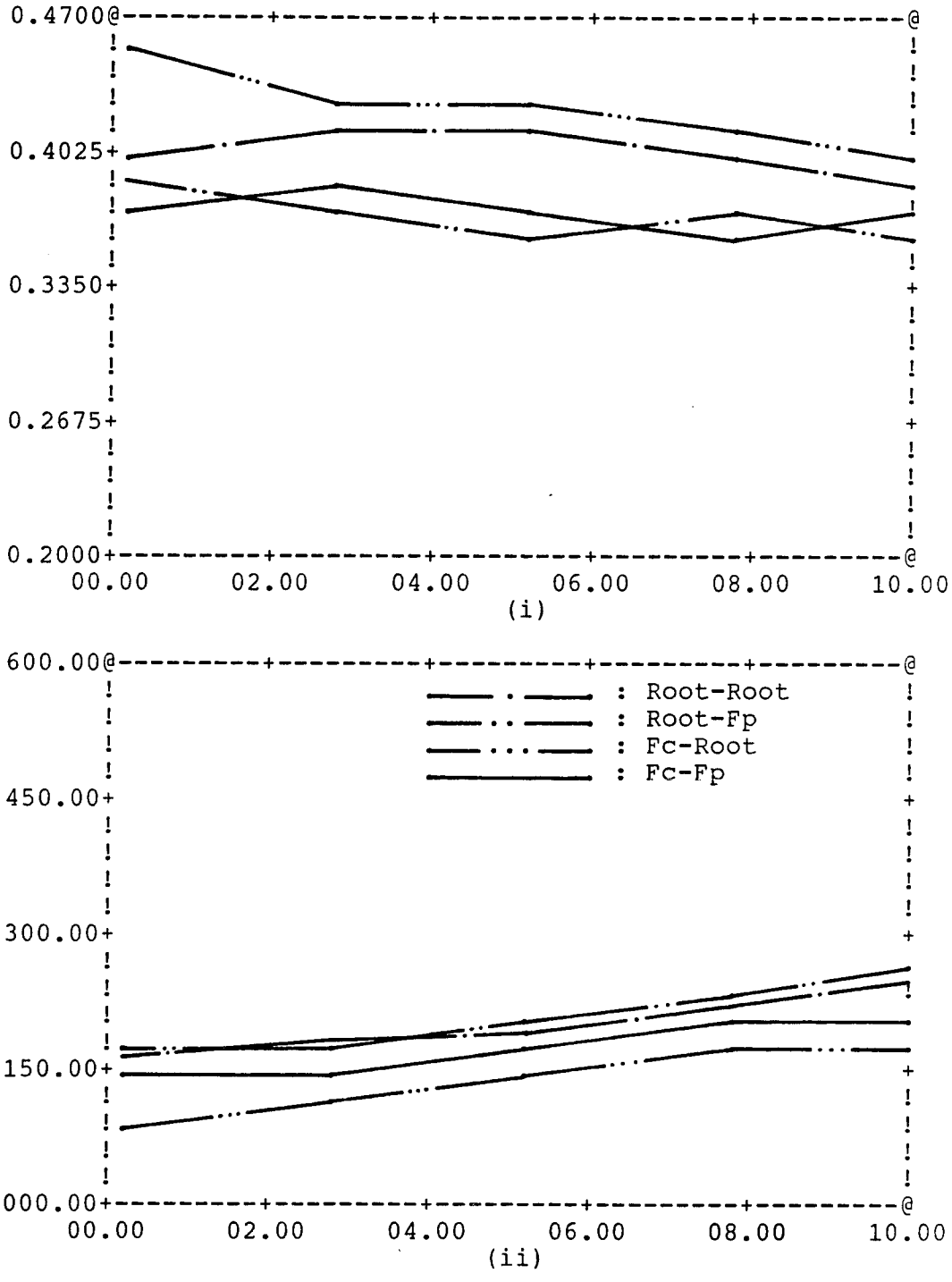


Figure 5.10 (i) Throughput vs CD and (ii) average TA vs CD of "multiserver" at fast arrival rate

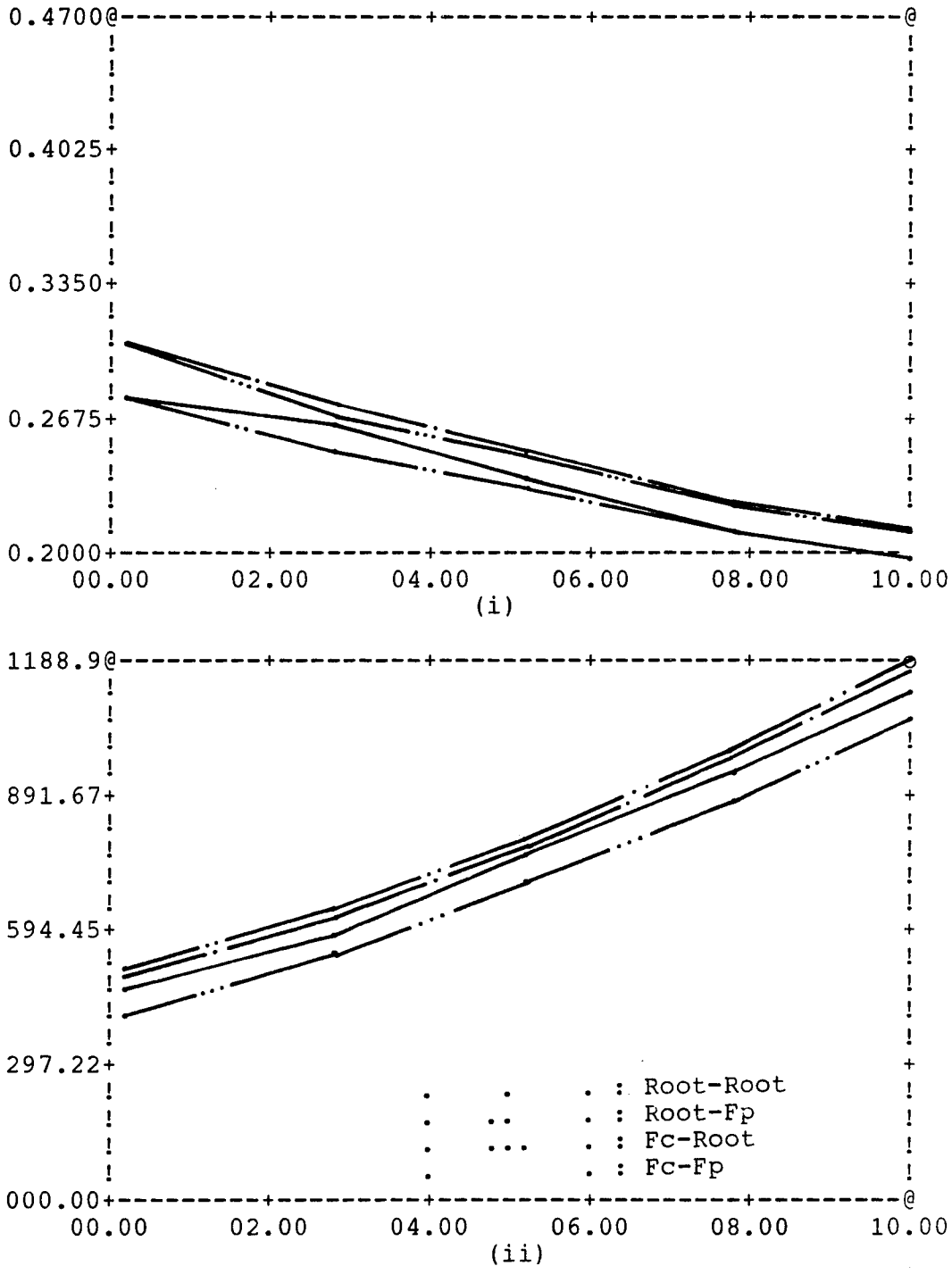


Figure 5.11 (i) Throughput vs CD and (ii) average TA vs CD of "pipeline" at fast arrival rate

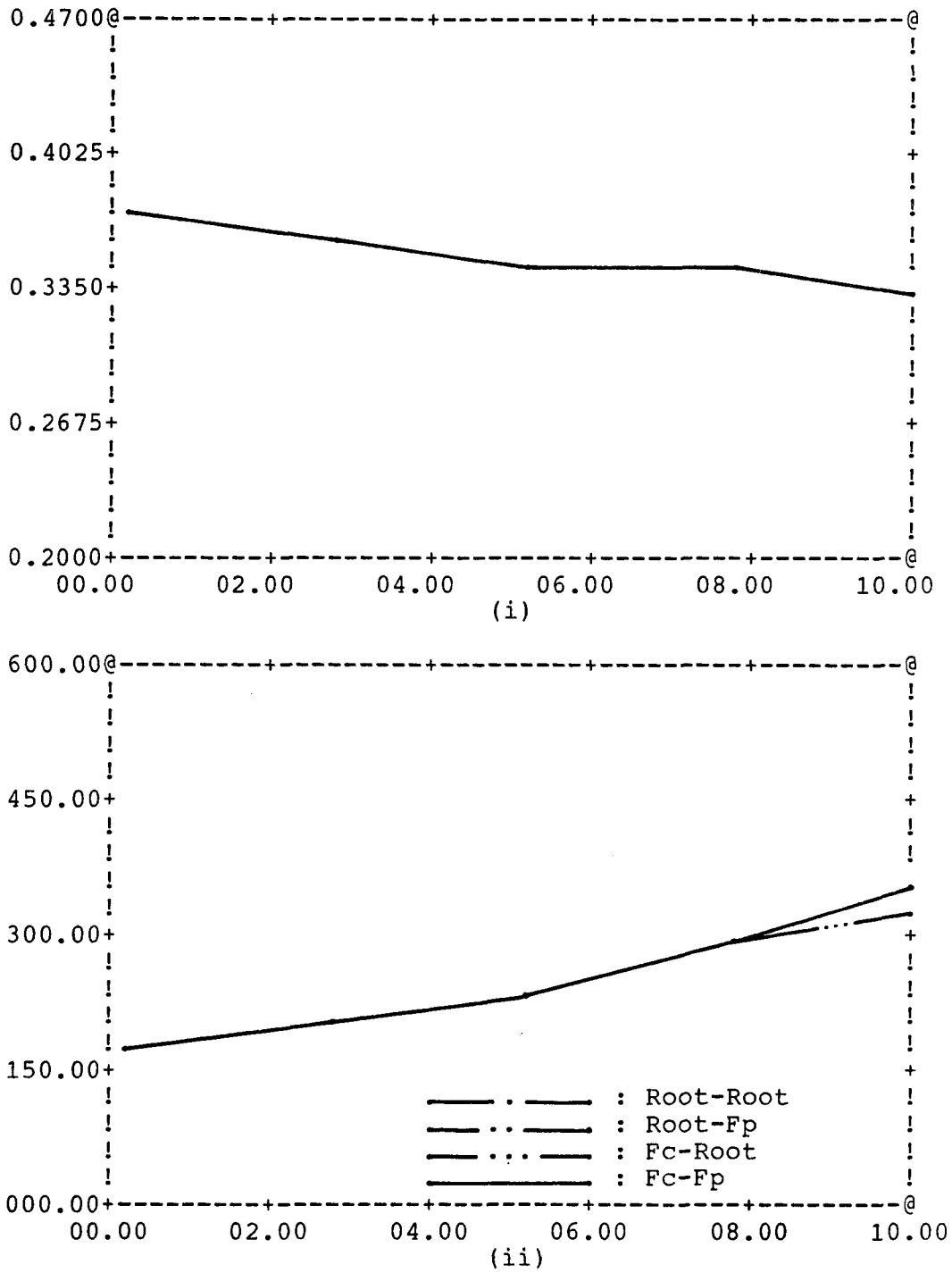
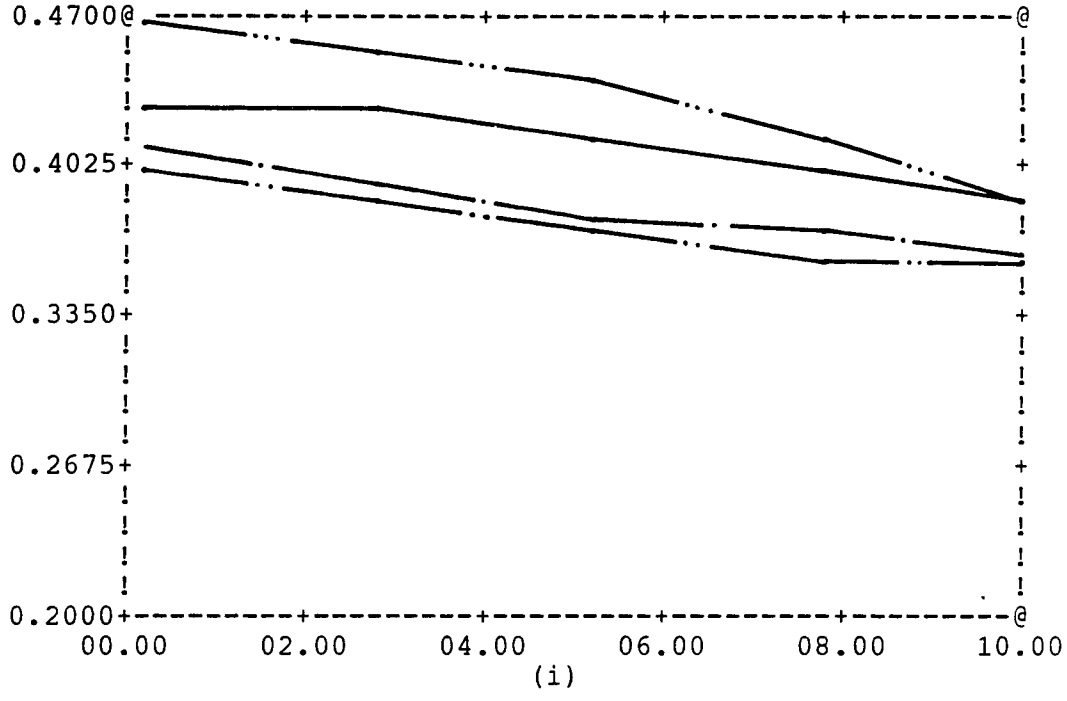


Figure 5.12 (i) Throughput vs CD and (ii) average TA vs CD of "divide'n'conquer" at fast arrival rate



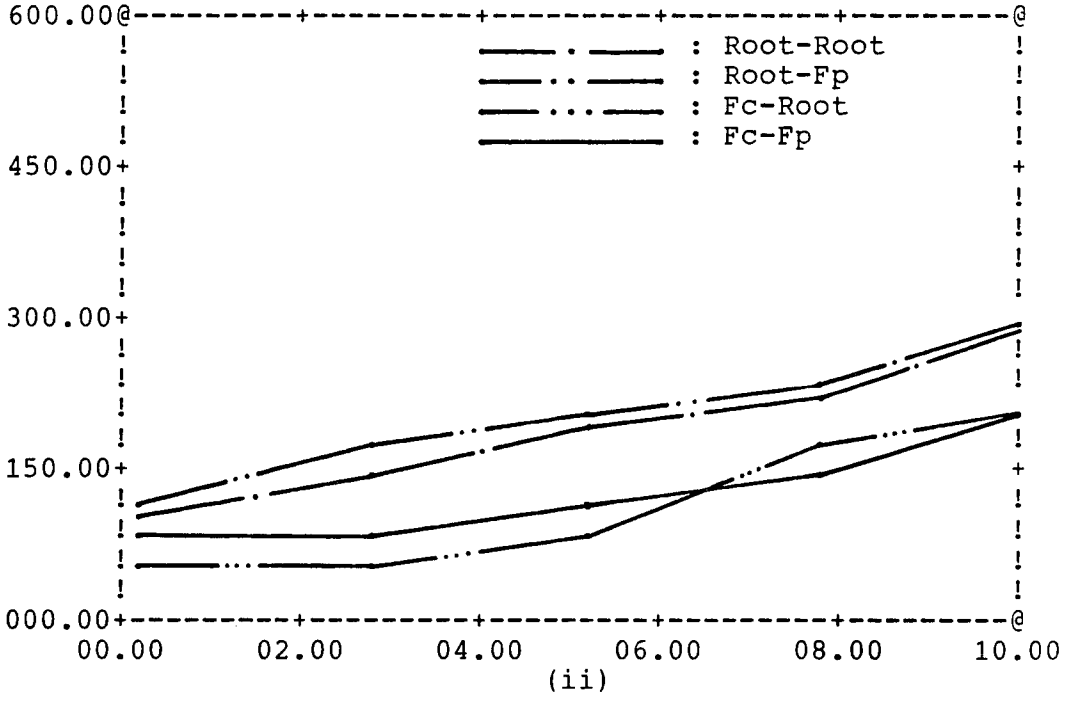


Figure 5.13 (i) Throughput vs CD and (ii) average TA vs CD of "combination" at fast arrival rate

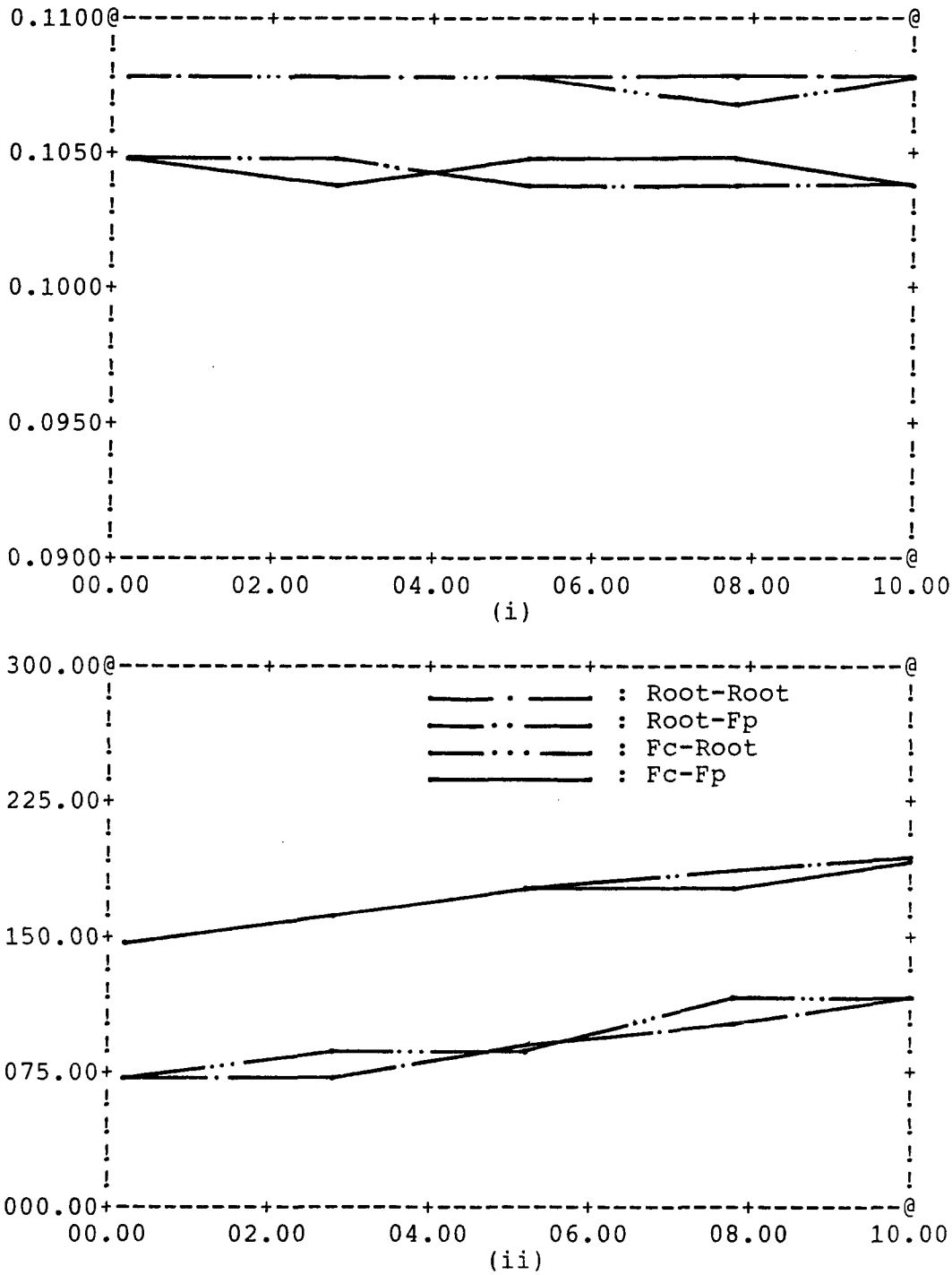


Figure 5.14 (i) Throughput vs CD and (ii) average TA vs CD of "multiserver" at slow arrival rate

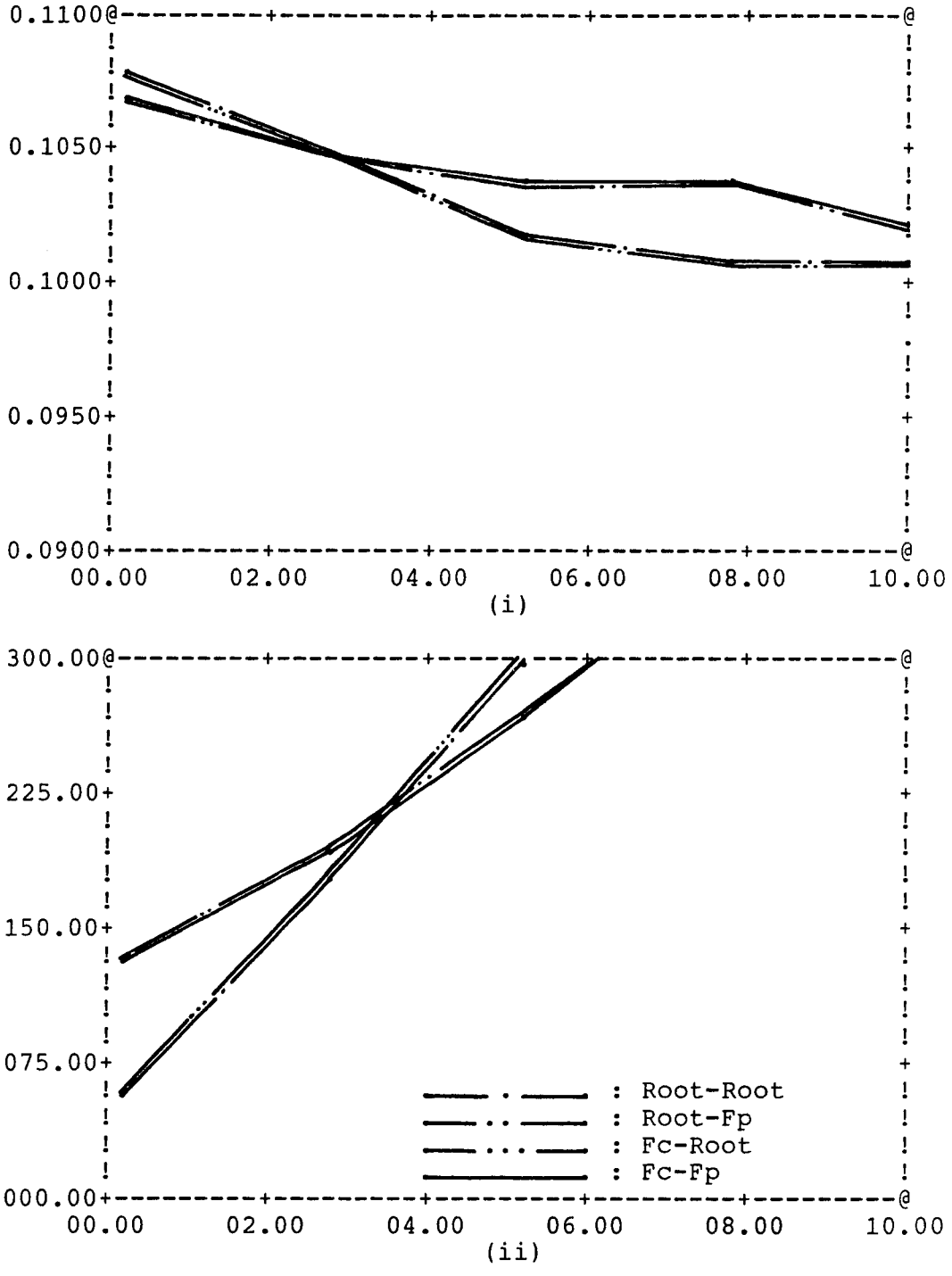


Figure 5.15 (i) Throughput vs CD and (ii) average TA vs CD of "pipeline" at slow arrival rate

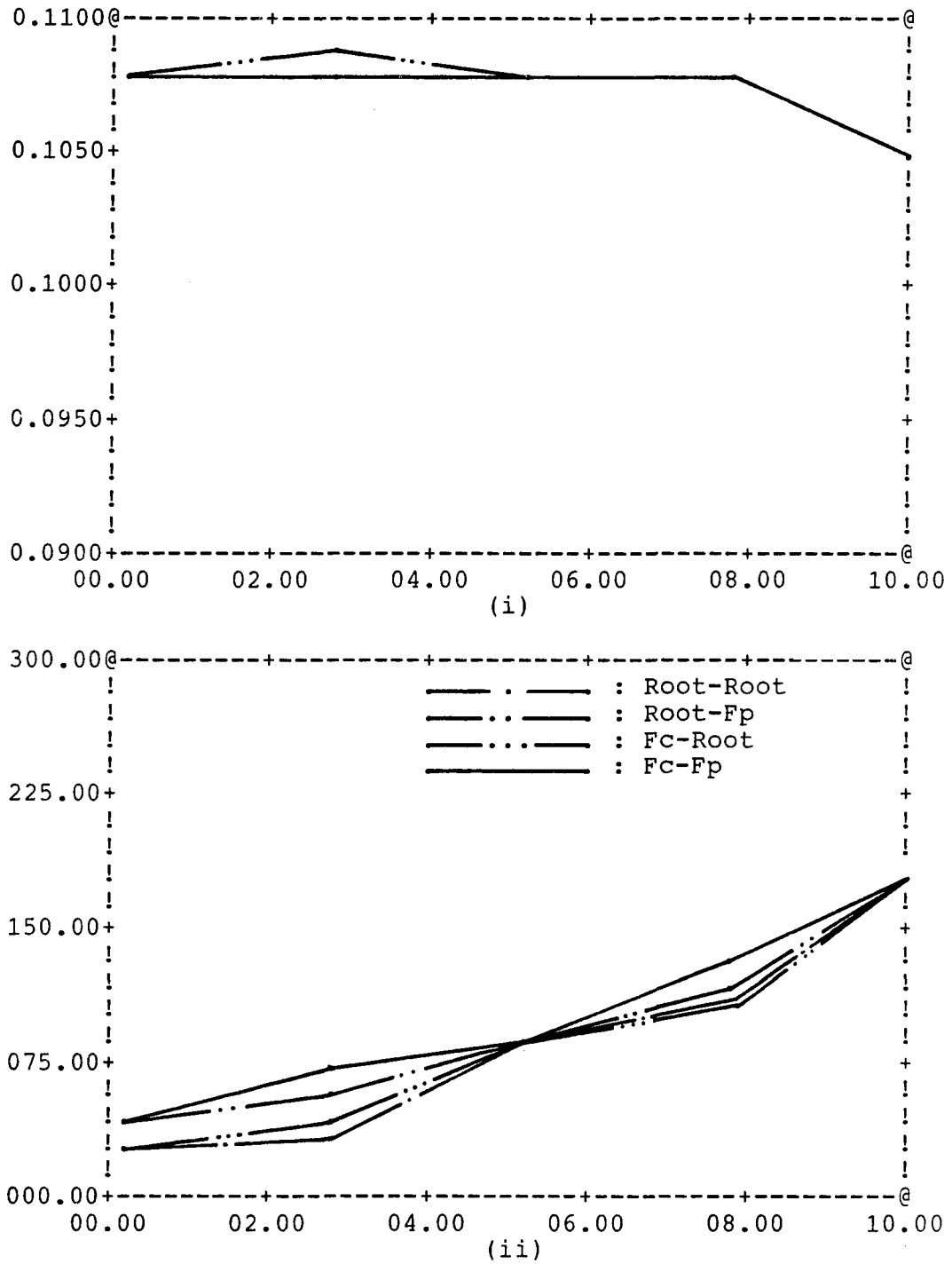
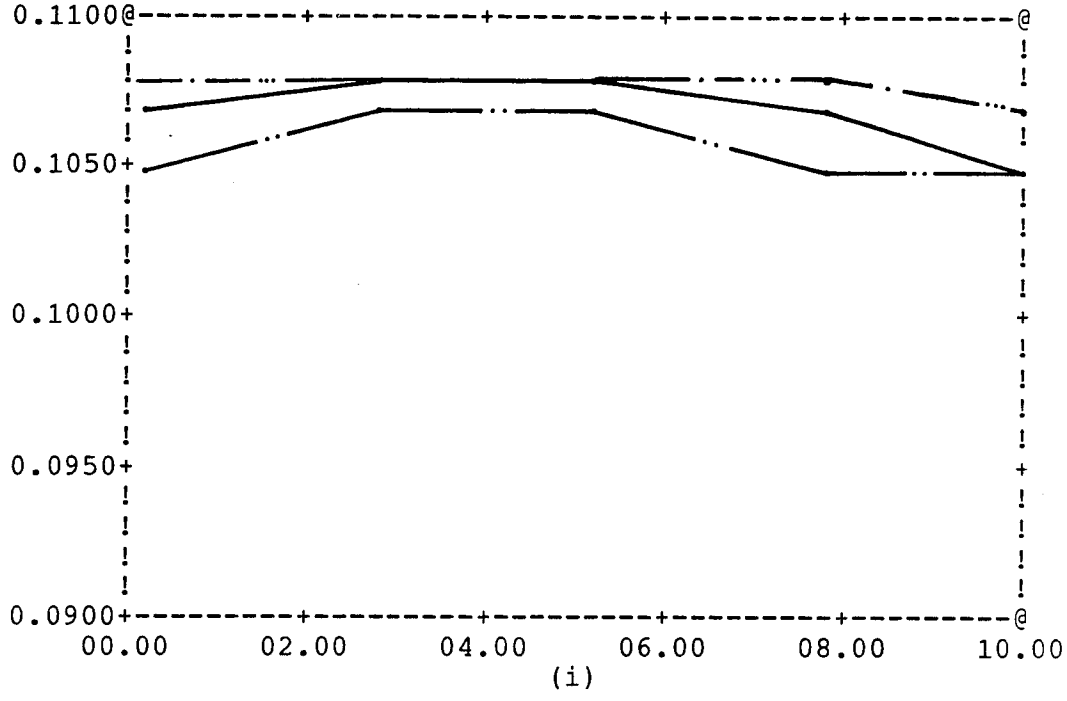


Figure 5.16 (i) Throughput vs CD and (ii) average TA vs CD of "divide'n'conquer" at slow arrival rate



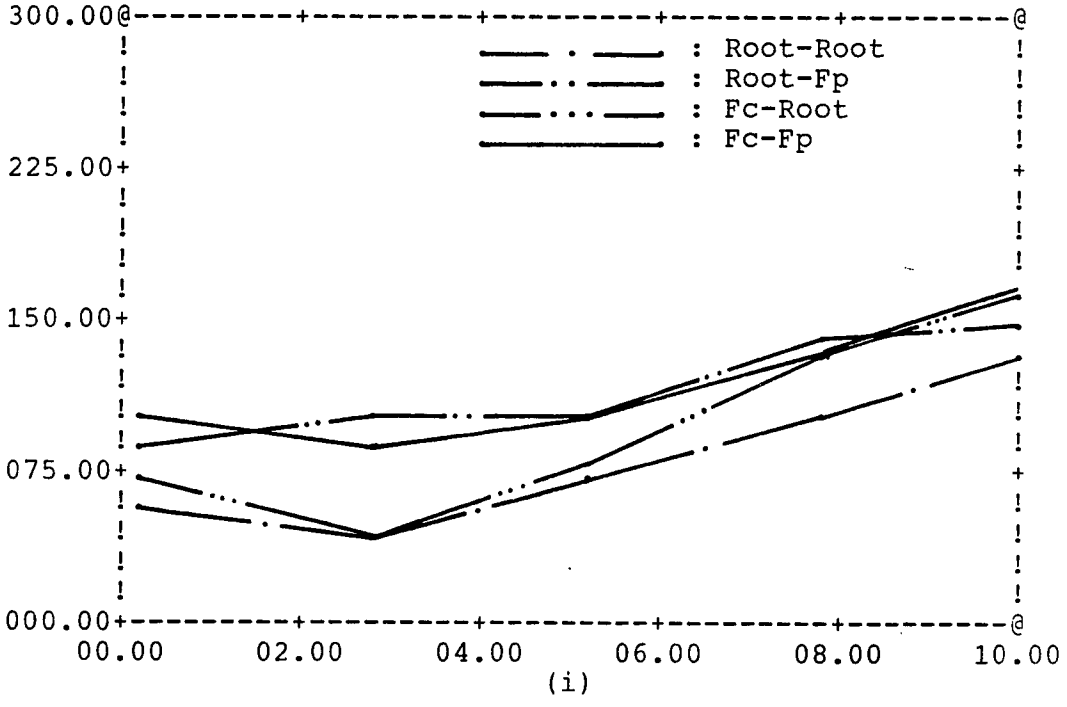


Figure 5.17 (i) Throughput vs CD and (ii) average TA vs CD of "combination" at slow arrival rate

Table 5.8 Comparisons of generation policies' performance at fast arrival rate

Coord	Gen Plcy		# best comparison CASE					# worst comparison CASE				
	Hire	Fire	1	2	3	4	5	1	2	3	4	5
MS	Root	Root										
	Root	Fp						o	oo	oo	o	oo
	Fc	Root	**	**	**	**	**					
	Fc	Fp						o			o	
PL	Root	Root	**	**	**	**	**					
	Root	Fp						oo	oo	oo	o	oo
	Fc	Root	**									
	Fc	Fp						oo	o		o	
DQ	Root	Root	**	*	*	*	*		o	o	o	o
	Root	Fp		*	*	*	*	oo	o	c	o	o
	Fc	Root	**	**	**	**	**					
	Fc	Fp		**	**	**	**	oo				
CO	Root	Root										
	Root	Fp						oo	oo	oo	oo	oo
	Fc	Root	**	**	**	*						
	Fc	Fp				*	**					

* : # of the best performance from turn-around time and throughput
o : # of the worst performance from turn-around time and throughput
Case 1: communication delay = 0.0 units
Case 2: communication delay = 2.5 units
Case 3: communication delay = 5.0 units
Case 4: communication delay = 7.5 units
Case 5: communication delay = 10.0 units

Table 5.9 Comparisons of generation polices' performance at slow arrival rate

Coord	Gen Plcy		# best comparison CASE					# worst comparison CASE				
	Hire	Fire	1	2	3	4	5	1	2	3	4	5
MS	Root	Root	*	**	*	**	*					
	Root	Fp						oo		oo	oo	oo
	Fc	Root	**		**		**					
	Fc	Fp						oo	oo		o	
PL	Root	Root	**	*	**	**	**					
	Root	Fp						oo	oo	o	o	o
	Fc	Root	**	**		*	*					
	Fc	Fp						oo	o	oo	oo	oo
DQ	Root	Root	**	*	**	*	**					
	Root	Fp	*			*	*	o	o	o		o
	Fc	Root	**	**	**	**	**					
	Fc	Fp	*			*	*	o	oo	oo	o	o
CO	Root	Root	**	**	**	**	**					
	Root	Fp						o	oo	oo	oo	o
	Fc	Root		*	*	*	*					
	Fc	Fp						o		o		oo

* : # of the best performance from turn-around time and throughput
o : # of the worst performance from turn-around time and throughput
Case 1: communication delay = 0.0 units
Case 2: communication delay = 2.5 units
Case 3: communication delay = 5.0 units
Case 4: communication delay = 7.5 units
Case 5: communication delay = 10.0 units

Referring to Table 5.1, the best policies shown above are steady in stable state and the worst policies are oscillating in the stable state. Therefore, we can conclude that the policies causing no oscillation of the number of processors at the stable state have better system performance and vice versa. The oscillation really slows down the system by repeatedly setting into "changing" mode and recovering to "normal" mode; once the processor is "changing" mode, packet processing is halted until it returns to "normal" mode.

5.5 Packet Batch Average Turn Around Time

The packet batch average turn around time (we will use "batch turn around time" for short) is measured the same as average turn around time except that the calculation of the average is done periodically instead of at the end of simulation.

In these experiments, we used "fast-to-slow" packet arrival rate and the arrival rate switching point is chosen long enough to allow the batch turn around time to reach a steady state. From experiments on batch turn around time, the packet turn around time against simulation time is studied and the behaviors of different coordinators and "hire/fire" generation policies are understood better. These empirical results are shown in Figure 5.18 to 5.21, in which communication delay is 10 time units. The curves are not of

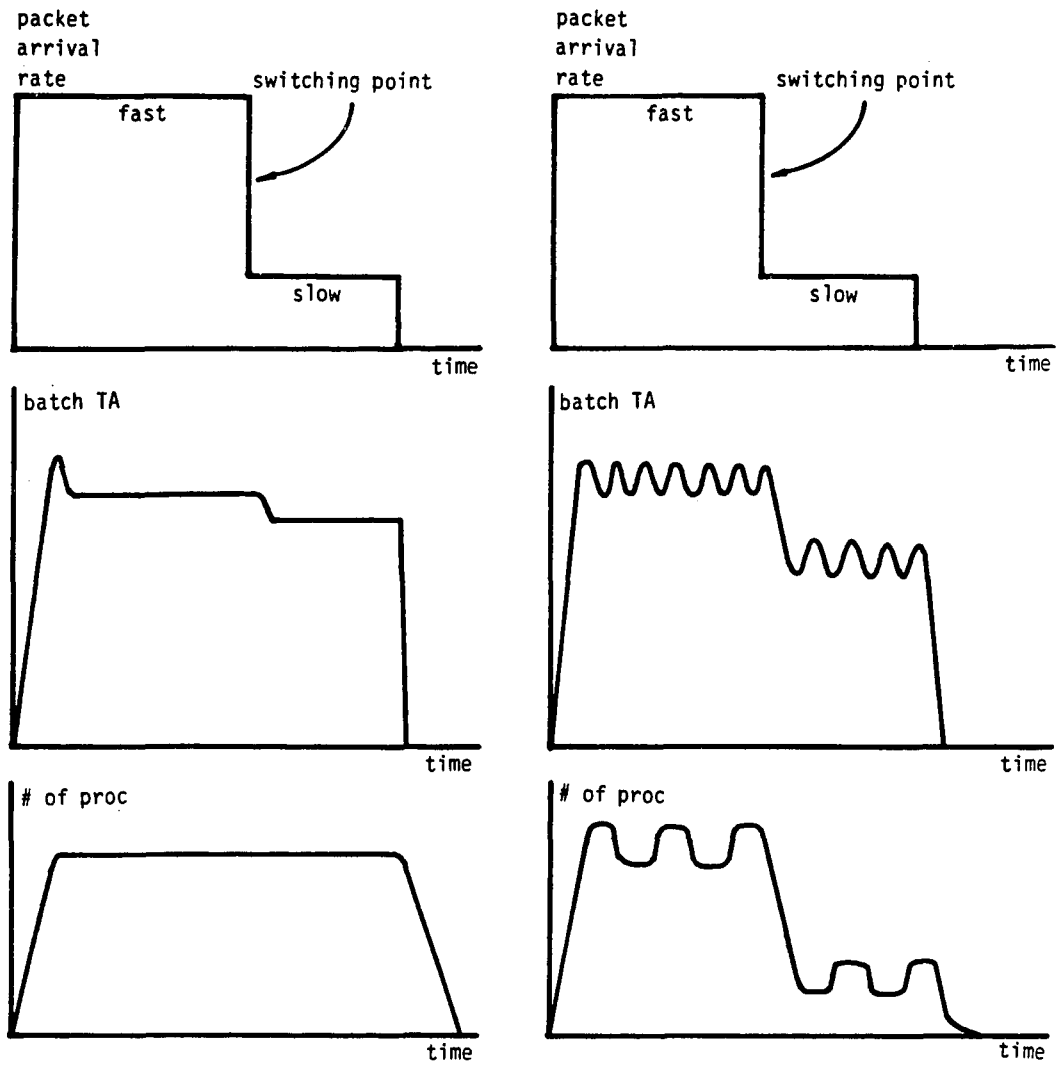


Figure 5.18 "Multiserver" in (i) "centralized" "fire" policy and (ii) "distributed" "fire" policy at "fast-to-slow" arrival rate

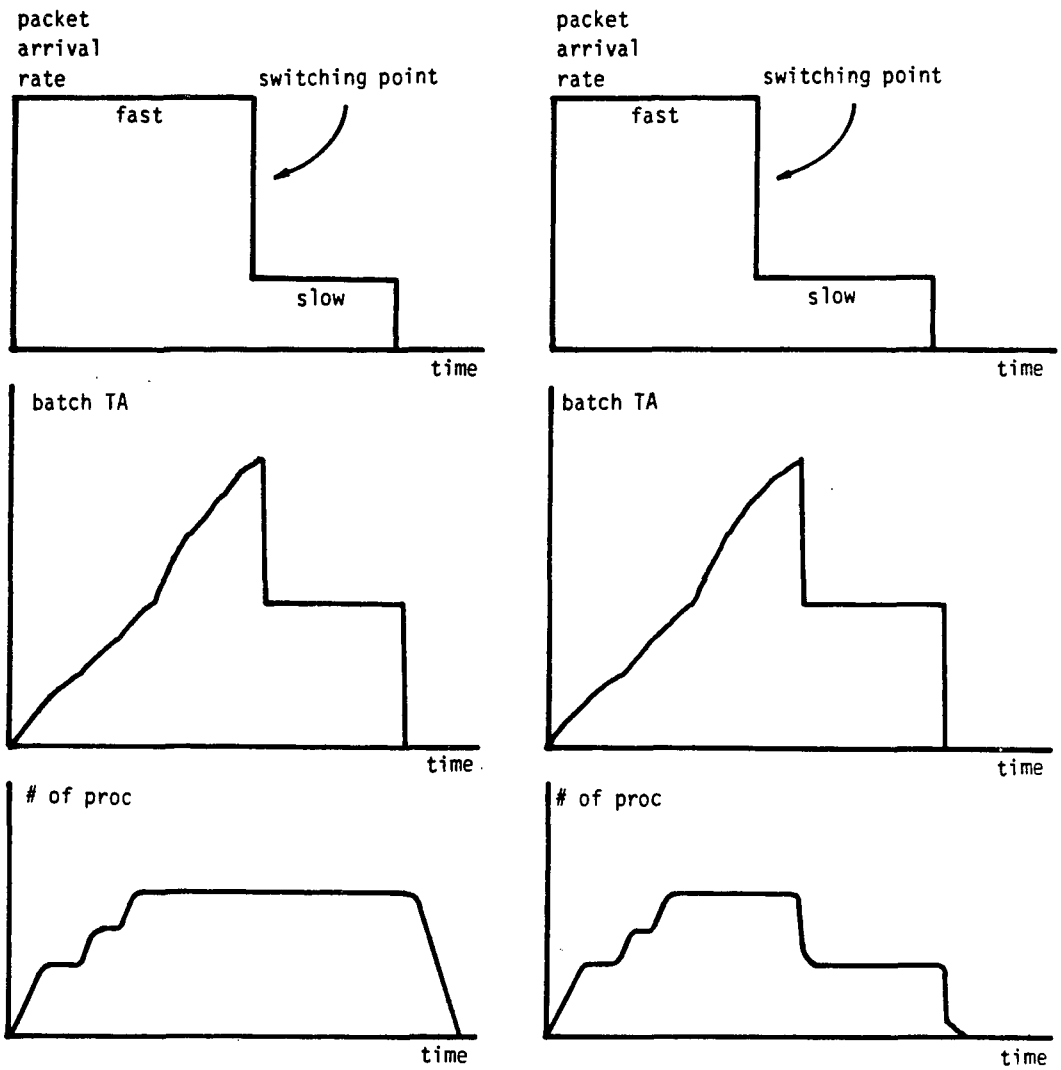


Figure 5.19 "Pipeline" in (i) "centralized" "fire" policy and (ii) "distributed" "fire" policy at "fast-to-slow" arrival rate

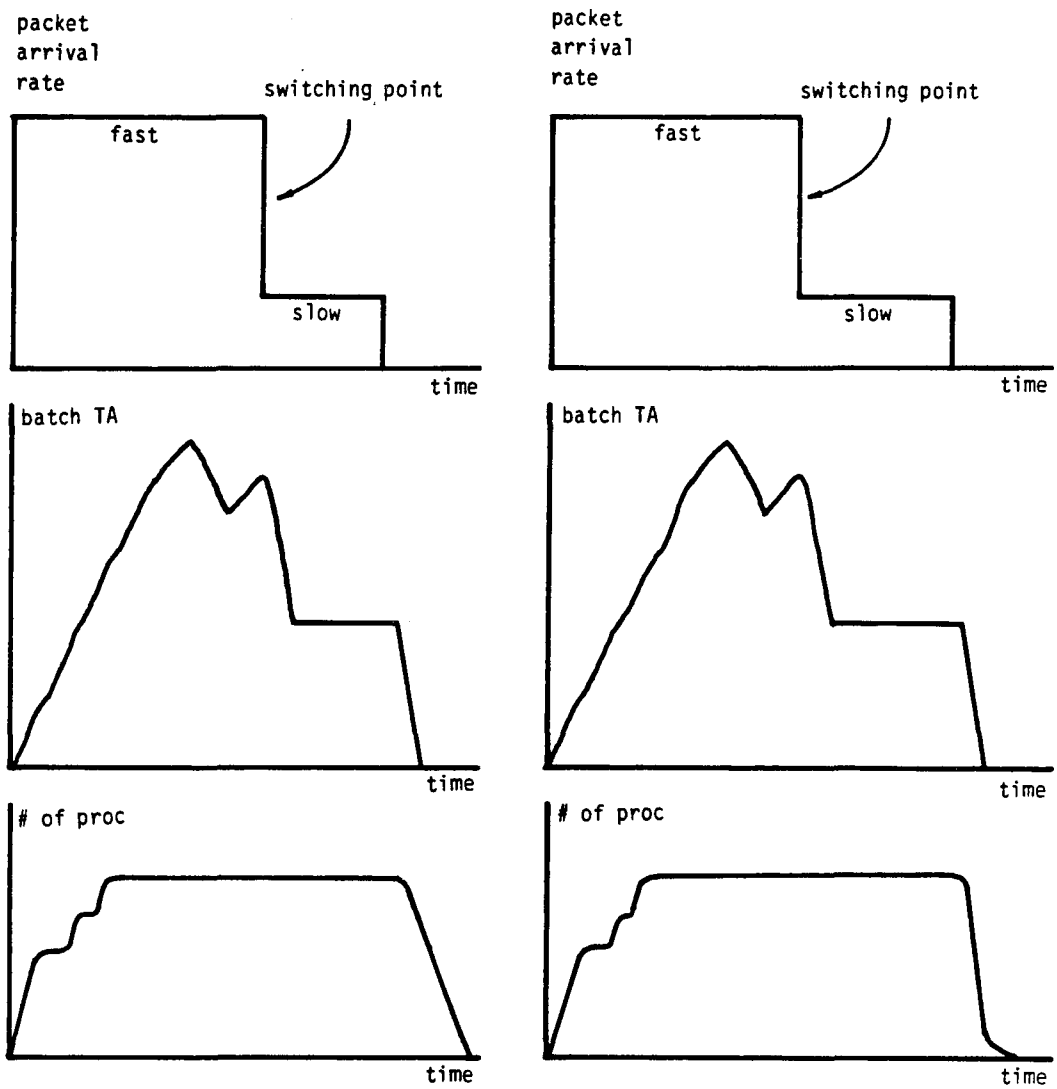


Figure 5.20 "Divide'n'conquer" in (i) "centralized" "fire" policy and (ii) "distributed" "fire" policy at "fast-to-slow" arrival rate

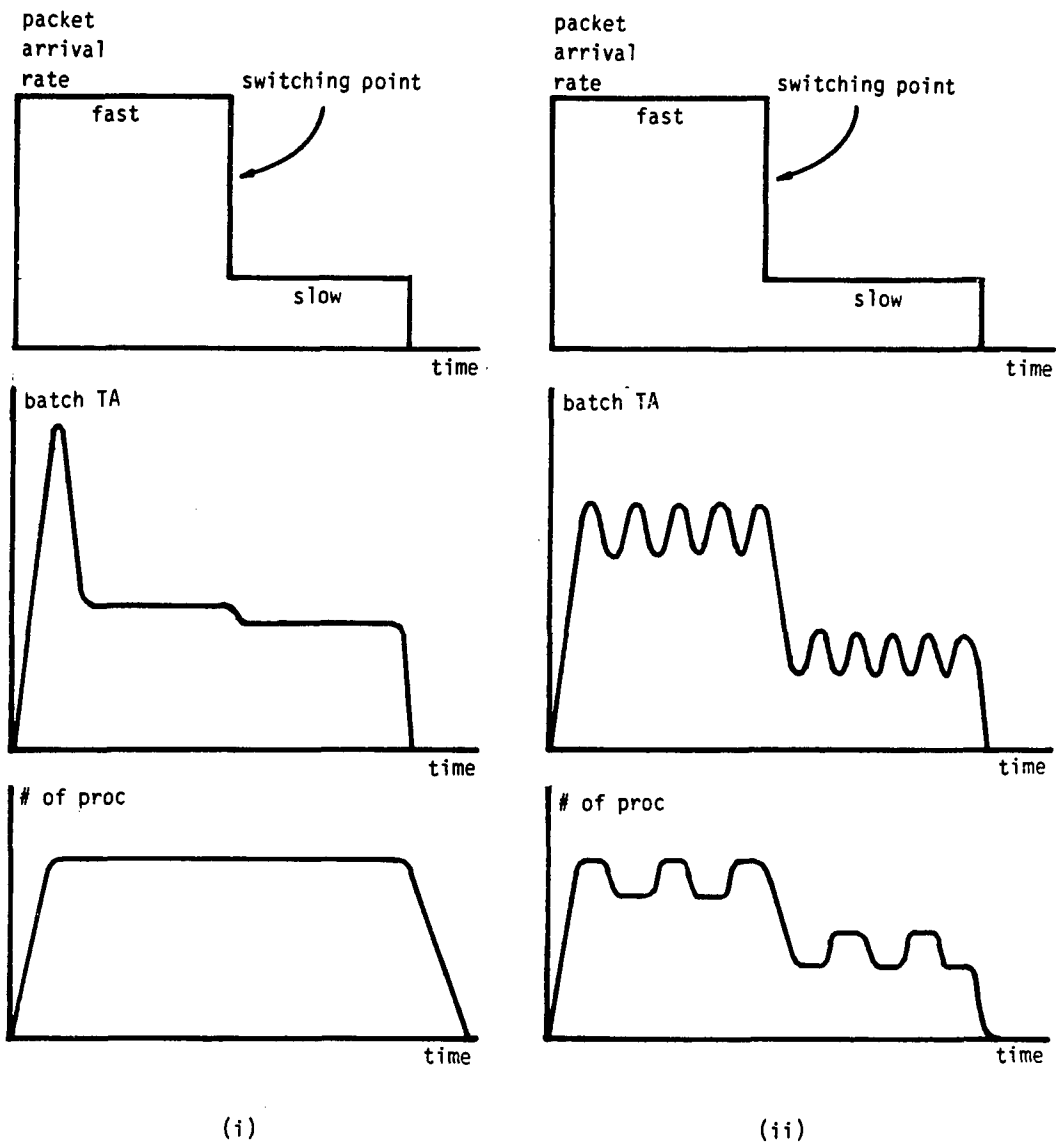


Figure 5.21 "Combination" in (i) "centralized" "fire" policy and (ii) "distributed" "fire" policy at "fast-to-slow" arrival rate

the same scale; no quantitative values of the experiments are shown.

The response of batch turn around time for each coordinator is different. For each coordinator type, we can see that the cases of centralized "fire" and distributed "fire" are also different. In the case of centralized "firing", the "multiserver" reaches a steady state, and, after the switching point, the batch turn around time only drops a little. In this case, the number of processors remains the same even though the packet arrival rate drops, and the packets suffer less waiting time in the BUFFER's queue (Figure 5.18 (i)). This phenomenon indicates that centralized "firing" is not able to adjust the number of processors of the system when the packet arrival has changed to slow rate. For distributed "firing", the number of processors oscillates, and so does the batch turn around time (Figure 5.18 (ii)).

In the case of "pipeline", batch turn around time seems never to reach a steady state in fast rate; it "blows up" (Figure 5.19 (i) and (ii)). This phenomenon also explains the steplike curve of the number of processors for the "pipeline". Since the packets are accumulated in the system, it has to "hire" more and more fp's whenever too many packets are in the system (Figure 5.1 (vi) and (vii)). However, the "pipeline" is able to reach steady state in slow rate. The number of processors remains the same in

centralized "firing" policy and drops to a lower level in distributed "firing" policy after the switching point.

In the case of the "divide'n'conquer", its batch turn around time also "blows up" but drops at a certain point and rises again (Figure 5.20 (i) and (ii)). It reaches steady state for slow arrival rate. The number of processors does not drop even through the arrival rate switches to slow. This can be interpretted that the total number of packets in the system still remains at a certain level because the incoming packets get divided again and again when they are sent level and level downward. In our cases, the "firing" policy is to compare the number of packets in the BUFFER's queue. Thus, the system does not "fire" away processors in this condition. Finally, the "combination" behaves very closely to the "multiserver" (Figure 5.21 (i) and (ii)).

Again, the "pipeline" shows the worst behavior in measuring and comparing the batch turn around time; it's batch turn around time can not reach a steady state in the fast packet arrival rate. The nature of a "pipeline" system tree is that it is usually not well balanced because the divided "lowerhalf" packets are always sent to the left child first, which, therefore, must "hire" more processors to relieve its workload; this results in a left "tilted" system tree. The more levels the tree has, the less the packet computation time is at the lower level. In this case,

the packets will be processed shortly at the lower level and returned to the upper levels. The "upperhalf" packets at the upper level still have longer computation time with comparison to the "lowerhalf" packets at the lower level. Therefore, packet bottle necks occur at the right children of the upper levels, in which they must "hire" more processors; the right children of the upper levels will be able to "hire" and the ones at the lower levels will not.

Moreover, there is a cut-off value [4] to determine whether a processor should stop "hiring" and to prevent the packets from being divided too small; the cut-off value is used to compare the last packet in the BUFFER's queue. If the range of last packet (u - l) is less the cut-off value, it will reject to carry out a "hire" operation. The cut-off value we used is 5.

In our experiments, the average packet computation time is about 50 time units. Therefore, the maximum number of the levels is approximately 4 or 5. The cut-off value also prevent unbalanced growth of "pipeline" trees, but this does not help to improve anything because the packets are taking more time in transmission (Figure 5.23 (ii)).

Let us consider the turn around time of a packet, which also reflects the nature of coordinators. We assume the packets have the longest computation time (in which $j > u > 1$) and the waiting time in the BUFFER's queue is

ignored. In a one level case (one "coordinator" and its children), they are (Figure 5.22):

. Multiserver:

    T + 2 * D

. Pipeline:

    T1 + T2 + 4 * D

. Divide'n'Conquer:

    Max(T1, T2) + Tc + 4 * D

where T: packet computation time.

T1, T2: computation time of the "lowerhalf" and the "upperhalf" packets (for j > u, T1 is equal to T2 and Max(T1, T2) = T / 2).

Tc: packet compiling time.

D: communication delay constant.

*: multiply.

In a "n" level tree, the packet turn around time of different coordinator is (Figure 5.23):

. Multiserver:

    T + n * 2 * D

    = T if D is small

    or

    = n * 2 * D if D is large

. Pipeline:

    (T1 + ... + Tm) + 4 * Xn * D

    = T if D is small

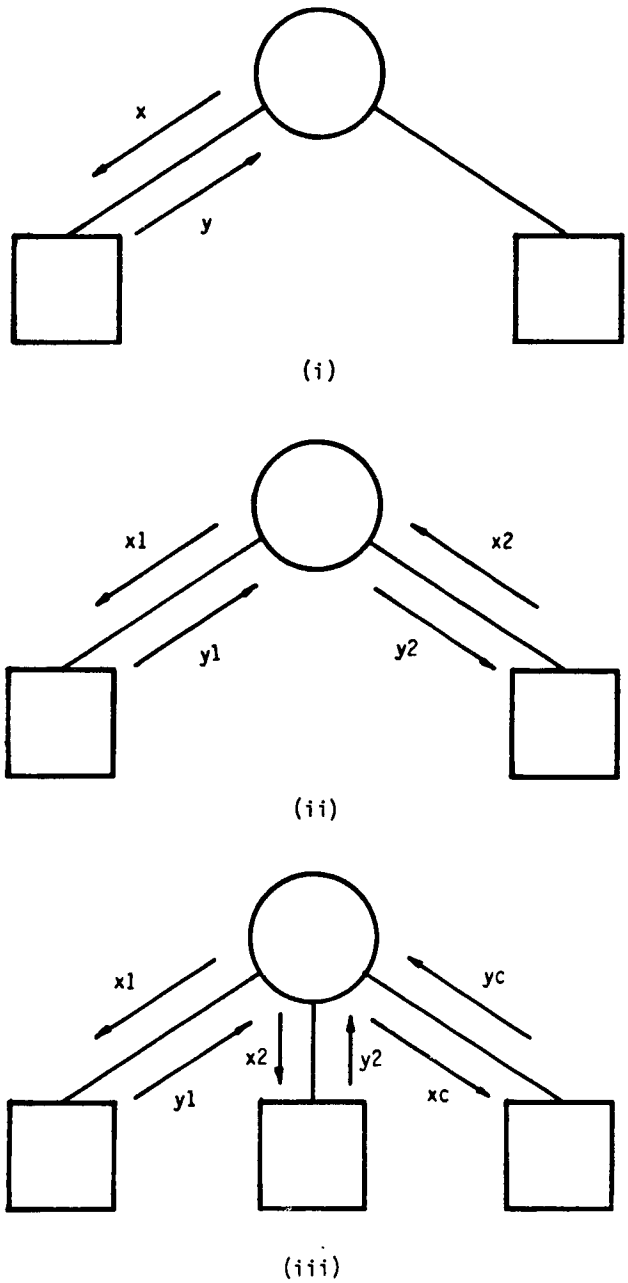
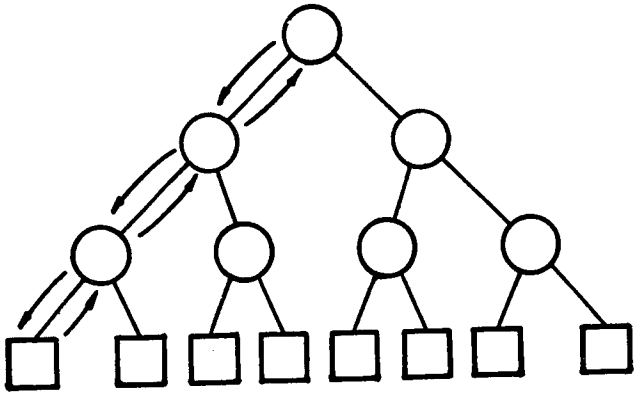
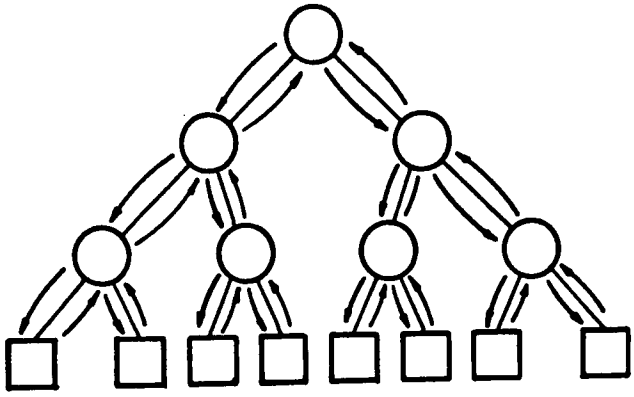


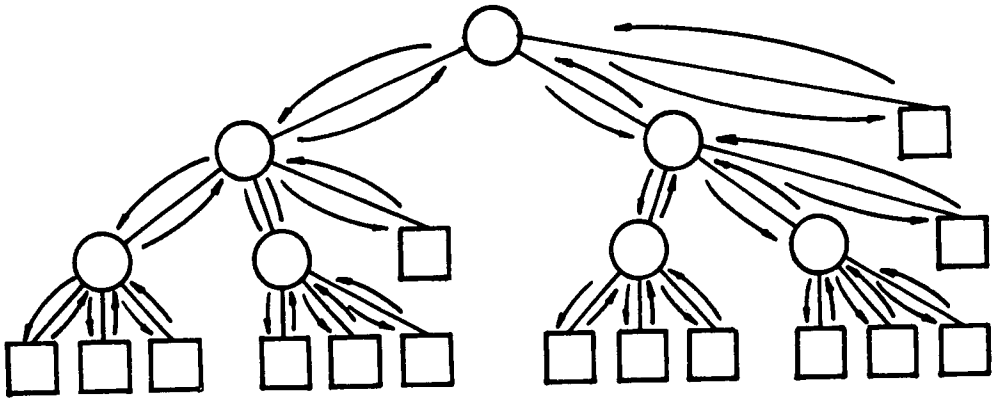
Figure 5.22 Packet path in a one level tree for (i) "multiserver" (ii) "pipeline" (iii) "divide and conquer"



(i)



(ii)



(iii)

Figure 5.23 Packet path in a three level tree for (i) "multiserver" (ii) "pipeline" (iii) "divide and conquer"

or

$= 4 * (2^n - 1) * D$ if D is large

. Divide'n'Conquer:

$Max(T1, ..., Tm) + n * Tc + n * 4 * D$

$= (T / 2^n) + n * Tc$ if D is small

or

$= n * 4 * D$ if D is large

where $Max(T1, ..., Tm)$ is equal to $T / 2^n$.

$Xn = sigma(1 + 2 + 4 + 8 + ... + 2^n-1) = (2^n - 1)$

In the slow packet arrival rate, these equations shown above reflects consistently with our simulation results. At the case of zero communication delay, the "divide'n'conquer" has the shortest turn around time, and the "multiserver" and the "pipeline" have approximately the same turn around time. The slight difference is due to the packet waiting time in the BUFFER's queue which is ignored previously. As the communication delay increases, the degree of influence in a descendent order is "pipeline", "divide'n'conquer", and "multiserver" because the "pipeline" has a larger factor $(2^n - 1)$ than the others. In the fast packet arrival rate, the results are different from the equations because packets suffer longer waiting time in the BUFFER's queue, which become a dominant factor to the turn around time and can not be ignored; our simplified equations

can not reflect the system reality.. However, the order of the degree of influence remains the same.

Let us define a simple performance ratio which measures the throughput of coordinators. If a factor is in favor of the system performance (for example, the number of packets can be processed at the same time), it is placed in numerator. Otherwise, it is put in denominator (such as the turn around time of a packet). The higher the ratio is, the better the system performs. The assumptions in previous discussion of packet turn around time sustain. The ratios in a one level case are (Figure 5.22):

. Multiserver: (in every (2 * t) time units, two packets can be processed and their turn around time)

$$\frac{2}{(T + 2 * D)} * \frac{1}{2 * t}$$

. Pipeline: (in every t time units, one packet can processed and its turn around time)

$$\frac{1}{(T1 + T2 + 4 * D)} * \frac{1}{t}$$

. Divide'n'Conquer: (in every t time units, one packet can be processed and its turn around time)

$$\frac{1}{(Max(T1, T2) + Tc + 4 * D)} * \frac{1}{t}$$

where t is the interarrival time of packet

As the tree levels grow, there are several advantages. First, the throughput is increased because more "f-computers" are available to process the packets. Second, the packet computation time is reduced by continuously dividing packets as they are transmitted level by level downward. Third, packets wait less time in the BUFFER's queue. The disadvantage is that the packets must suffer more communication time while being transmitted from one level to another. To obtain the best system performance, there must exist an optimum relationship between the number of levels and the communication delay. Similarly, for a "n" level tree, the ratios are (Figure 5.23):

. Multiserver: (in every (2^n * t) time units, 2^n packets can be processed and their turn around time)

$$\frac{2^n}{(T + n * 2 * D)} * \frac{1}{2^n * t}$$

$$= \frac{1}{T * t} \qquad \text{if D is small}$$

or

$$= \frac{2^n}{n * 2 * D * t} \qquad \text{if D is large}$$

. Pipeline: (in every t time units, one packet can be processed and its turn around time)

$$\frac{1}{(T1 + \ldots + Tm) + 4 * Xn * D} * \frac{1}{t}$$

$$= \frac{1}{T + 4 * Xn * D} * \frac{1}{t}$$

$$= \frac{1}{T * t} \quad \text{if D is small}$$

or

$$= \frac{1}{4 * Xn * D * t} \quad \text{if D is large}$$

. Divide'n'Conquer: (in every t time units, one packet can be processed and its turn around time)

$$\frac{1}{(Max(T1, \ldots, Tm) + n * Tc + n * 4 * D)} * \frac{1}{t}$$

$$= \frac{2^n}{T + (n * Tc + n * 4 * D) * 2^n} * \frac{1}{t}$$

$$= \frac{2^n}{(T + n * Tc * 2^n) * t} \quad \text{if D is small}$$

$$= \frac{2^n}{T * t} \quad \text{Tc is small in our model}$$

or

$$= \frac{1}{n * 4 * D * t} \quad \text{if D is large}$$

In the slow packet arrival rate, they are consistent with our simulation results. At the zero communication delay, the "divide'n'conquer" has the highest throughput. As the communication delay increases, the degree of influence in a descendent order is "pipeline", "divide'n'conquer", and

"multiserver" because the "pipeline" has a larger factor (2^n - 1) at its denominator. In the fast packet arrival rate, the packets' waiting time in the BUFFER's queue becomes a dominant factor and can not be ignored; our simplified ratios we have shown can not reflect the reality. However, the order of the degree of influence remains the same.

As a matter of fact, the best system performance depends on an optimum relationship between the number of the tree levels and the communication delay; the advantages and the disadvantages of the growth of the tree reach a compromise. It is possible that the "combination" of "multiserver" and "divide'n'conquer" gives the best system performance. For instance, we assume that, in a specific environment, the "divide'n'conquer" probably works best as it is a three level tree and the "multiserver" does best as being a two level tree. A system use a "multiserver" as its root which, then, uses two three level "divide'n'conquer" trees as its children may result in best system performance.

Besides, our experimental results showed that the number of levels of processors were different from one type of coordinator to another. Recall the maximum number of processors used in the system in Table 5.1 and we can figure out the depth of the tree. For example, in the slow packet arrival rate, the "multiserver", the "pipeline", and the "divide'n'conquer" have about 3, 4, and 4 levels respectively. In the fast packet arrival rate, all of them

have about 5 levels. Even through, the packets in the "pipeline" tree suffer much more transmission time and result in a worse performance.

CHAPTER 6

CONCLUSIONS AND FUTURE APPLICATIONS

In our work, we extended Linos' model by introducing a "channel" module to handle the communication delay between processors. Furthermore, over 800 simulation replications were run and the results were obtained by running each experiment five times and averaging those data.

6.1 Conclusions

The system's behaviors vary greatly with different configurations and in different environments. In "hire/fire" operations, the number of processors alters much faster in the "distributed" generation policy than in the "centralized" generation policy; the systems using "distributed" policies switch much faster subjected to the environment changing. However, the systems using "distributed" "fire" policy tend to oscillate instead of being in steady state. It is not economic to do that for two reasons. First, some processors in the system are "jumping" between the "changing" and the "normal" modes; the packet processing is slowed down because the packet processing is halted in "changing" state. Second, it is time consuming and unrealistic to allocate and deallocate processors frequently in real hardware devices. This is very much related to the

transmission and acceptance policy criteria we used based on the number of packets in the BUFFER's queue. Some other criteria such as average packet turn around time of each processor [3] and average packet computation time of each fp [1] have been used.

Moreover, the "pipeline" is proven to be worst and should not be considered to be implemented in the real case. Combination using the "multiserver" and the "divide'n'conquer" gives best results in the fast packet arrival rate, and the "divide'n'conquer" is best in the slow packet arrival rate. In the comparisons of the policies, in most cases, the "Fc-root" is proven best and the "root-Fp" is worst. We also found that the policies causing no oscillation of the number of processors at stable state have better performance and vice versa; this is consistent with the discussion mentioned earlier.

6.2 Future Researches

There are several fields in our model are open for future researches and worthy further studies. We can mention as follows:

1) New policies and criteria which generate, transmit, accept, or execute the adaptive control signals: the policies in which "fire" signals generated by "f-parents" tend to cause oscillations of the number of processors at steady state. For instance, broadcasting the adaptive

control signals instead of transmitting them may reduce the oscillations.

2) New policies that are sensitive to the relationship of the levels of system tree and communication delay: the system should stop "hiring" as the advantage of reducing the packets' computation time by dividing them ("divide'n'conquer") or increasing the system throughput by "hiring" more processors ("multiserver") turns to be a disadvantage because packets are taking too much time in transmission.
3) Policies or system configurations giving the best results in some specific environments: for example, the "multiserver" works best in the high communication delay environments and the "divide'n'conquer" is the best in the environment of low communication delay and slow packet arrival rate.
4) New algorithms, configurations, and structures for the modules of flexible processor: modifications of modules or packet handling algorithms can be applied to improve the overall performance.
5) New type problems which are related to the real world applications: in this model, we applied the problem of linear search, and, for example, we can use quick-sort algorithm to handle the sorting problems, which is so far under development by our architecture group.

6.3 Future Applications

There are two major fields of applications -- the new computer system development and the robot organization of a space station [4]. Designers are able to choose the special coordinator types and system configuration to meet their needs such as throughput and turn around time according to the preformance experimental results we have obtained.

REFERENCES

1. Kim, T. G.
"Experiments and Discussion on Performance of Hierarchically Coordinated Adaptive Architectures ECE (574 Project Report)," University of Arizona, 1985.

2. Law, A. M.
"Simulation of Manufacturing Systems," Class Notes of MIS 521B, University of Arizona, 1986.

3. Liaw, Y. S.
"An Experimental Frame for a Modular Hierarchically Coordinated Adaptive Computer Architecture," Master Thesis, University of Arizona, 1986.

4. Linos, P. K.
"Simulation of a Modular Hierarchically Coordinated Adaptive Architecture," Master Thesis, Wayne State University, Michigan, 1985.

5. Russel, E. C.
"Building Simulation Models with SIMSCRIPT II.5", CACI, L. A., 1985.

6. Zeigler, B. P.
Class Notes of ECE 574, University of Arizona, 1985.

7. Zeigler, B. P. and Reynolds, R. G.
"Towards a Theory of Adaptive Computer Architectures," Proceedings, The 5th Internaltional Conference on Distributed Computing Systems, Dencer, Colorado. Computer Society Press, 1985, 468-475.