Efficient Message-Based System for Concurrent Simulation

by

Moses Hsingwen Ma

S.B., Massachusetts Institute of Technology (1981)
S.M., Massachusetts Institute of Technology (1981)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1989

Signature of Author ....[illegible]............
Department of Electrical Engineering and Computer Science
January 11, 1989

Certified by ....[illegible]............
Robert H. Halstead Jr.
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....[illegible]............
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Efficient Message-Based System for Concurrent Simulation

by
Moses Hsingwen Ma

Submitted to the Department of Electrical Engineering and Computer Science
on January 11, 1989, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis describes and analyzes a concurrent discrete-event simulation system based on Jefferson's Time Warp system[8]. A simulation system consists of a simulator and a simulation. The simulation is decomposed into tasks that communicate through messages. The messages dictate the amount of parallelism possible. Without messages, tasks could run independently of each other. With messages, however, one task may not be able to start until another task is complete. Each message contains a virtual receive time which indicates the order in which the message should be processed (by the receiving task) relative to other messages. Tasks send and receive messages asynchronously with respect to one another; it is therefore possible for a task to receive a message with a virtual receive time lower than that of a message it has already processed. When this occurs, the task must rolled back and its previous state restored. Such an event is called a ***backup***.

System performance is defined in terms of maximizing parallelism and minimizing backups. We present, analyze and verify a model that describes the underlying phenomena that govern the performance of a simulation.

We explore three aspects of system design in the quest for optimizing performance:

- How tasks are scheduled to run on physical processors.
- How to partition a simulation up into groups of tasks in order to avoid contention for task queues.
- How to implement the backup process.

For each one of these design aspects, several strategies are described and analyzed.

Thesis Supervisor: Robert H. Halstead Jr.
Title: Associate Professor of Computer Science and Engineering

Acknowledgments

I would like to first thank all the people who made this thesis possible (for without their help all of this would never have been completed).

I would like to thank Mike Athans, who allowed me to look for a second chance.

I would like to thank Dan Nussbaum who recommended me to Bert Halstead and the Real-Time Systems (RTS) group.

I would like to thanks the people of RTS. My stay in RTS was probably the most pleasant of my career at MIT. The people in RTS sure know how to take care of their own. Thanks again to Sharon Thomas.

I would like to especially thank my thesis supervisor, Bert, for giving me my second chance. He also provided great inspiration and encouragement which made this thesis possible. Without Bert's patience and sharp mind this thesis would never have been possible. Everything Bert pushed for just made the thesis so much better in the end.

I would like to thank Margaret St. Pierre, who encouraged me to continue when I was about to quit about three years ago.

I would like to thank Juan Loaiza, Randy Osborne, Pete Osler, Morry Katz, and Laura Bagnall for putting up with me as an officemate.

I would like to thank Juan Loaiza and Dan Nussbaum for discussing major issues of my thesis. Even though Juan thinks no one will ever read this and Dan is totally unreliable.

I would like to thank the two undergraduates who helped me with my thesis. They were David Rho and Ethan Rappaport. Dave helped automate the simulation runs and made statistic gathering easy.

I would like to thank my thesis readers. They were Chris Terman and John Tsitsiklis. John was my officemate during my first thesis attempt.

I would like to thank all of the people in the MIT Bridge Club. No one really knows what would have happen if I did not get involved with the MIT Bridge Club. Without the Bridge Club, I might have finished my thesis five years earlier. Then again without the Bridge Club, I could have gone crazy with a nervous breakdown. I think there is a tradeoff there. All in all I think the Bridge Club taught me some very important lessons.

Finally I would like to thank Melisse Leib. Melisse has really made finishing this thesis worthwhile. Without her constant encouragement and support this thesis would never have been completed. Her love, affection and friendship provided much of the impetus that was required to complete this thesis.

I would also like to thank Melisse for discussing major issues of my thesis. She also kept her half of our deal to help proofread my thesis. She spent many late nights helping me rewrite major portions of my thesis. I think I got the better part of the deal. I only had to type her thesis, term paper and resume.

This research was supported in part by the Defense Advanced Research Projects Agency and was monitored by the Office of Naval Research under contract numbers N00014-83-K-0125 and N00014-84-K-0099.

I dedicate this thesis to my love, Melisse.

Table of Contents

I. INTRODUCTION

1.1. Problem Statement

As we enter an age when fast sequential processors are reaching their limits, one approach to increasing computational power is the use of multiprocessors. In the past, simulations executed sequentially could take weeks to execute on even the fastest and most efficient of modern computers. However, in many simulation applications, concurrent processing can be used to hasten this execution. This thesis addresses the issues involved in implementing simulations using concurrent processing.

The approach followed here uses the messages passed between processes to dictate the amount of parallelism possible. Inter-process communication is the impediment preventing any concurrent simulation from being neatly decoupled, and hence from being run cleanly in parallel. Our simulation system is implemented in an object oriented programming style with message passing; the system is written in Multilisp, a concurrent dialect of Lisp developed by Halstead[7].

We faced many important design decisions. First, how should processing elements be mapped to the simulation's tasks? Second, how should these processing elements "move" to other tasks within the simulation? Third, how should the simulation be partitioned into groups of tasks? Fourth, should we allow tasks to progress forward aggressively, thus incurring the risk of backup, or only conservatively, thus avoiding backup? Fifth, how should a general and user friendly simulation system be built? Sixth, how should we define a clear user interface? Seventh, what were the important parameters affecting simulation performance?

1.1.1 Justification

In the future, multiprocessors may consist of thousands of individual processors. Inter-processor communications will be much improved. Such a machine will have become problem limited, rather than machine limited. That is, its limits will be defined by how well designed its mechanism is for taking advantage of the problem's inherent parallelism. Three major approaches strive to develop this mechanism: parallel algorithms, distributed systems, and concurrent simulations. Our approach is via concurrent simulation and is essentially concerned with how well the problem simulation's correlated tasks can be juxtaposed. The aim of this thesis is to upgrade existing simulation mechanisms, taking advantage of the simulation problem's intrinsic parallelism.

Essentially, the contribution of this thesis is to implement an efficient message-based system for concurrent simulation with well defined interfaces, and to understand the effects of the different parameters on simulation performance.

1.2. Literature Survey

1.2.1 Introduction

Simulation systems provide means for observing the behavior of large systems over time when direct observation of the system is inconvenient or impossible or when it is desirable to view the system behavior at an increased or decreased rate of time passage relative to the observer.

Two separate classes of simulation exist: discrete and continuous. In discrete simulations, changes in the state of the system take place at discrete points in time and are instantaneous. In continuous simulations, changes in state occur smoothly and continuously in time. An example of a discrete system is a queue at a bank teller. An example of a continuous system is a global weather system. Numerical analysis and analog computing are used to solve systems of differential equations that model the behavior of continuous systems. In contrast, event-based simulation is often used for discrete systems.

Our primary motivation for attempting to perform simulation in a distributed manner comes from the observation that some interesting classes of systems exhibit a high degree of natural parallelism; the system can be divided into tasks that execute in parallel.

In concurrent simulation, the simulation is decomposed into tasks that are simulated in a distributed manner over a network of processors by assigning each task to a processor. This approach is attractive for the simulation of network models because of the inherent parallelism, and because of their widespread application to computer systems and communication networks. Such parallelism can be exploited in the decomposition to give a potentially more cost-effective method of simulation. The distributed approach, however, requires the proper synchronization of the components for the simulation to be carried out correctly. One way synchronization can be achieved is by message passing between the tasks.

An upper bound on the parallelism available is determined by the number of tasks into which the simulated system can be decomposed. We wish to analyze various methods for achieving either the lowest execution time for a given simulation, or the most cost-effective simulation. Because of the overhead introduced by distributing the simulation, the lowest execution time may not necessarily coincide with the largest degree of parallelism. The amount of overhead introduced depends on the relative speeds of generating and transferring messages between processes, and also on the particular method of synchronizing the processes.

In this approach, the actions of these processors are synchronized, while each processor is responsible for processing the events related to its set of tasks. This distributed approach can reduce the total time necessary to perform a given simulation. This accomplishment depends upon the number of processors available. If a sufficient number of processors exists, then a one-to-one mapping of tasks to processors is possible. However, a one-to-one mapping may not result in the fastest execution, or may be too expensive. This is because some tasks may execute slower than others, potentially causing faster tasks to wait or backup, depending on the nature of the simulation. This problem can be alleviated by making a many-to-one mapping from tasks to processors so that the work assigned to each processor is about equal.

1.2.1.1 Taxonomy

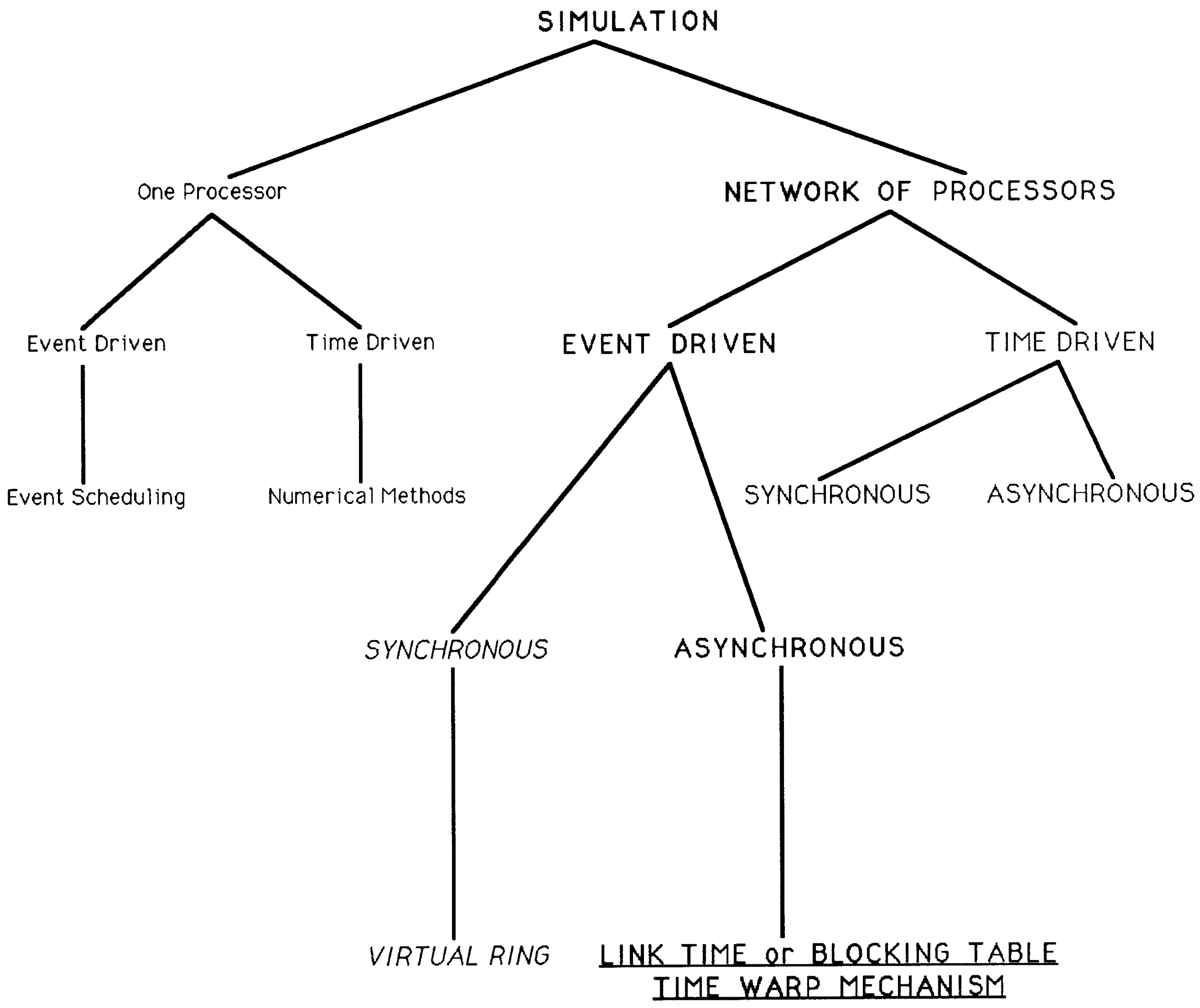


Figure 1.1 Taxonomy Tree for Simulation Systems

We define a taxonomy as shown in Figure **1.1**, which classifies various synchronization methods. At the first level we distinguish whether there is one or a network of processors available. In a network, the simulation is decomposed into tasks and distributed over the processors. The next level deals with the nature of the simulation. In event-driven simulation the changes in the system state are simulated only when events occur, and the sequence of virtual times associated with a sequence of events is monotonically increasing. In time-driven simulation a universal virtual clock is incremented by a fixed amount for each step in a simulation. All of the changes in one interval of a system state are simulated before advancing the virtual time to the next interval. A fundamental feature of algorithms for the synchronization of time-driven simulations is that a central controller must broadcast a signal to every simulation task to indicate the end of a simulation interval.

In the multiprocessor case, we have a third level, dependent on the value of virtual time at each node. Many methods have been proposed for implementing concurrent or distributed simulation. They can be broadly classified into two groups, synchronous and asynchronous.

In a synchronous distributed simulation, all tasks progress forward in virtual time simultaneously. The usual event queue implementations of sequential simulation are all synchronous methods. In contrast, an asynchronous distributed simulation permits some tasks to run ahead of others in virtual time. Of course, an asynchronous simulation must include some mechanism for ensuring that no events are executed in the wrong order when a task that is behind schedules an event for execution by a task that is ahead. Within this constraint, an asynchronous method tries to maximize the number of events being executed in parallel. We emphasize that the difference between synchronous and asynchronous methods is in the implementation. As there is no semantic difference, it will be invisible to the simulation programmer.

Algorithms for event-driven simulation on a network of processors, developed by Chandy and Misra [5,6], and Bryant [4], are the virtual ring algorithm for synchronous event-driven simulation and the link-time and blocking table algorithms for asynchronous event-driven simulation.

1.2.1.2 Concurrent Simulation

Discrete event-driven simulation differs semantically from other computational paradigms primarily because of the notion of virtual time, which plays a special logical role in the global coordination of the computation. A simulation on one processor is normally organized as a collection of procedures, coroutines, or pseudo-parallel processes invoked according to a special scheduling discipline, executing the task with the lowest virtual time first.

The following sections discuss two different methods that have been employed to perform concurrent simulation: one a conservative method and the other aggressive. For this thesis, the latter strategy is implemented because of various problems that were found in the conservative strategy. Since many implementation issues were neither designed nor fully specified in the original studies of the aggressive strategy [13], a major portion of this project concerned itself with many design issues such as system structure, scheduling, interface and user specifications. These sections provide an overview of the Network Paradigm (conservative) and the Time Warp (aggressive) methods for concurrent event-driven simulation, covering most of the previous work done on concurrent simulation.

1.2.2 Network Paradigm and Conservative Mechanisms

Several groups have been active in the field of asynchronous distributed simulation. Two of these are Chandy and Misra [5,6] at the University of Texas at Austin, and Peacock, Wong, and Manning [16,17,18] at the University of Waterloo. Their work falls into the Network Paradigm category; they include the Link Time and Blocking Table algorithms of Peacock, Wong and Manning; and the Chandy and Misra method.

In the Network Paradigm, each task in the simulation is represented as a deterministic sequential process acting as a node in a network. Communication channels are represented by directed arcs between the nodes of the network. Each task has associated with it a local virtual time which is the virtual time of the last event executed by that task. Local virtual time is a measure of how far the process has progressed in the simulation. Communication between tasks in the network (called events, customers, transactions, etc.) consists of time-stamped messages that traverse the arcs between the nodes of the tasks. Each message has a time-stamp that specifies the virtual time for processing the message. Also each message's time stamp must be greater than or equal to the sender's local virtual time at the moment of sending.

A task in the Network Paradigm may have several input channels each of which maintains separate input message queues. In the Network Paradigm it is assumed that each of the network's arcs preserves the order of the messages, and messages sent along any particular arc must form a non-decreasing sequence in terms of their time-stamps. Since the queues are ordered by increasing time-stamps, determining the next message to process requires examining only the first message in each queue. Therefore, each node can easily merge its several input message streams by selecting the lowest time-stamped message off one of its input message queues. The previous requirements guarantee that no message can arrive later with an earlier time-stamp than the selected message.

Each task continuously executes the following algorithm:

1. Waits for each input message queue to have at least one message enqueued in it.
2. Begins processing by taking the message with the lowest time stamp off one of the input message queues.
3. Updates the local virtual time for this node to be the value of the message's time-stamp.
4. Performs the actions dictated by the message. This could involve changing the node's local variables and/or sending one or more time-stamped messages along the output arcs of the node to other tasks.
5. Go to step 1.

Even if each of the n nodes in the simulation were assigned to different processors and all could execute concurrently, optimal n-fold speedup over the usual single processor case is not usually possible because severe problems can occur, preventing such ideal behavior from happening in *real* simulations: the first is the limited amount of concurrency available, and the second, is the inherent cost of concurrent or distributed computation. The first limitation is that the simulation must take at least as long as the most time consuming task. The second limitation is due to the extra time needed for synchronization.

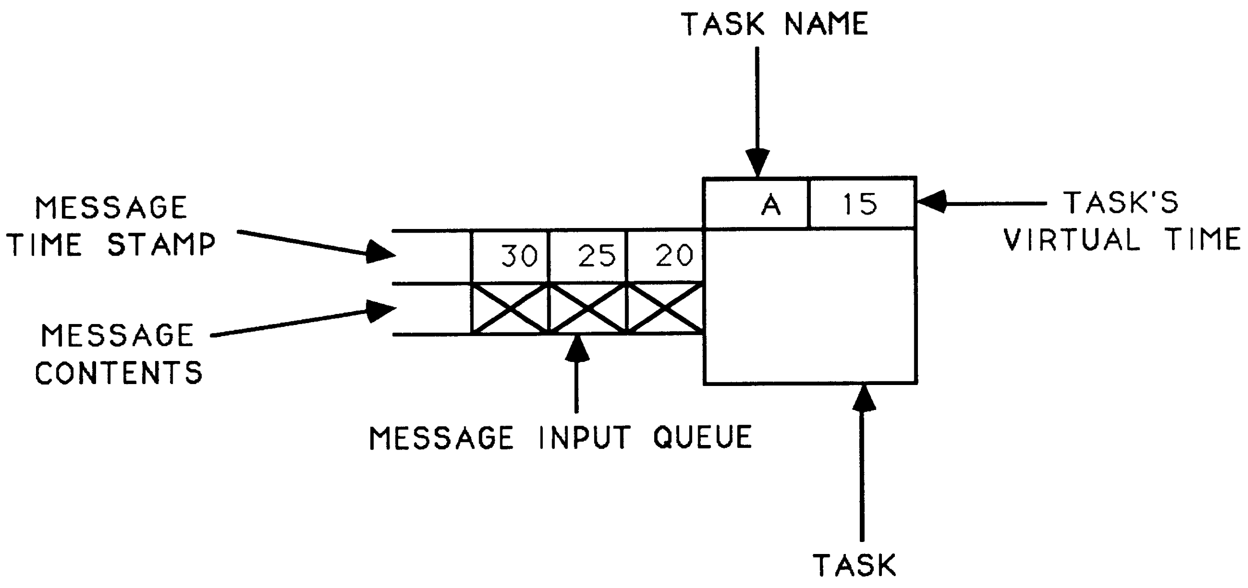


Figure 1.2 Key to Diagrams

Two problems are associated with Step 1 of the algorithm. The problem arises when a node has one or more of its input queues empty when it tries to receive the *next message*. If this occurs, then the node must wait, because if it accepts a message from any of the nonempty input queues, it has no guarantee that later it will not receive, along one of the currently empty input arcs, a message with a time-stamp earlier than the message it will have processed. If the node were to process a message from a non-empty arc and later received a message with an earlier time-stamp from a previously empty arc, this would constitute a logical error in the simulation because events would be simulated in the wrong time order. For example in Figure **1.3**, task **D** must block because one of its input arcs has an empty message queue.

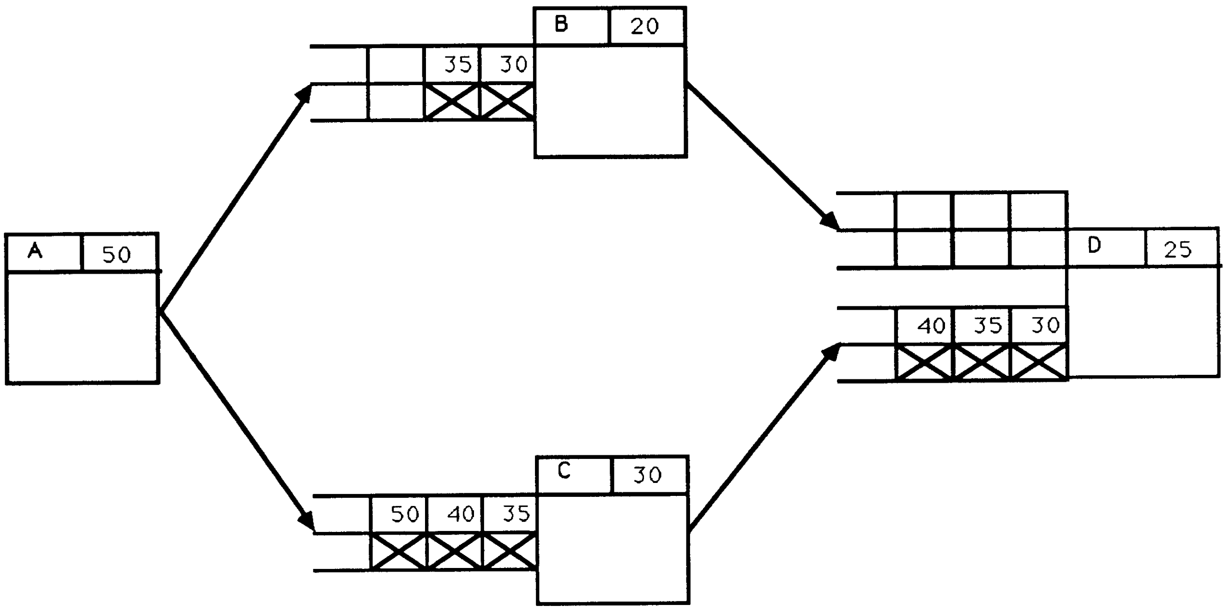


Figure 1.3 Conservative Mechanism within the Network Paradigm

A task is safe at a particular moment if it has at least one message queued on each input arc. A task with at least one empty input arc is unsafe. A conservative mechanism for concurrent simulation is one in which at any given moment, all safe tasks are considered eligible to execute, but unsafe tasks are suspended until they are safe. Any conservative mechanism will be input/output equivalent to the usual sequential event-list simulation mechanism, if we assume that both will terminate without memory overflow, runtime error or deadlock; each task will receive the same sequence of messages as the sequential mechanism, will progress through the same sequence of states, will send out the same messages to other tasks, and will produce the same final output. Any mechanism that is input/output equivalent will return the same set of outputs given the same set of inputs.

The best possible case for the conservative strategy is shown in Figure 1.4. In the example, the simulation is of a network with no fan-in or directed cycles, where each node has only one input queue. In this case each node is eligible to execute whenever it has at least one unprocessed message in that queue; therefore, deadlock is impossible in this special case. Since no unnecessary blocking occurs, the degree of concurrency is

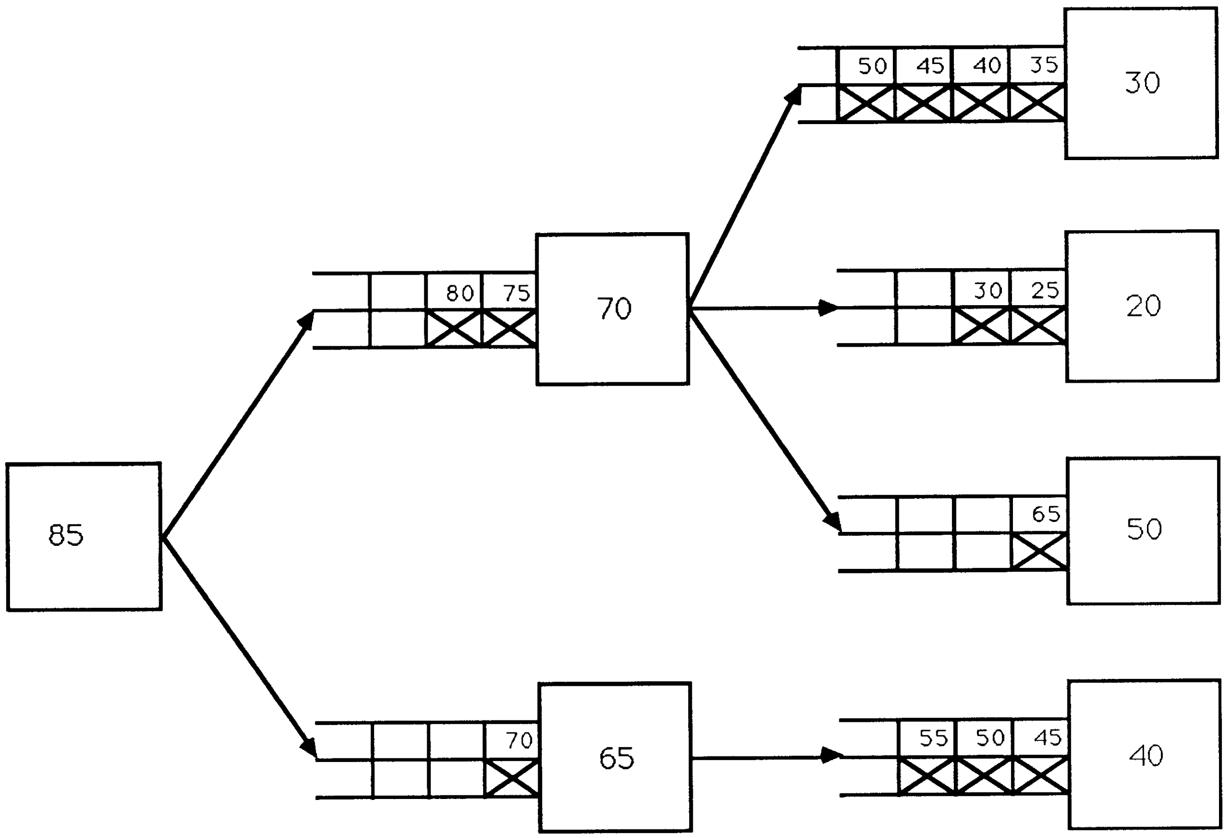


Figure 1.4 Network where the Conservative Mechanism works well

maximal.

Conservative mechanisms are unsatisfactory since for most simulations they may encounter memory overflow or deadlock. Even in the absence of these problems, conservative mechanisms allow for only limited concurrency in most simulations.

A second problem is illustrated in Figure **1.3**. If the rate at which **B** and **C** together produce messages exceeds the rate at which **D** can accept messages, then one or both of **D**'s input queues will eventually overflow with messages. This is a flow control problem that troubles all distributed systems. However, the problem with the conservative mechanism is that if **B** produces messages at a much slower rate than does **C**, then most of the time **D**'s input queue of messages from **B** will be empty, and **D** will be blocked. Meanwhile, messages from **C** can build up and will eventually overflow memory, even though **D** may be intrinsically fast enough to process the messages at a rate higher than the sum of the production rates of **B** and **C**. The problem is that **D** will spend most of its time blocked. We may assume that **B**'s virtual time is advancing more slowly than **C**'s, because if **B**'s virtual time were keeping up with **C**'s, just one message from **B** would *unblock* a large number of messages from **C** to **D**. In many simulations it is impossible to predict whether this blockage will occur. The likelihood is influenced by the variations in timing, by the details of process scheduling, and by the nondeterministic nature of the simulation.

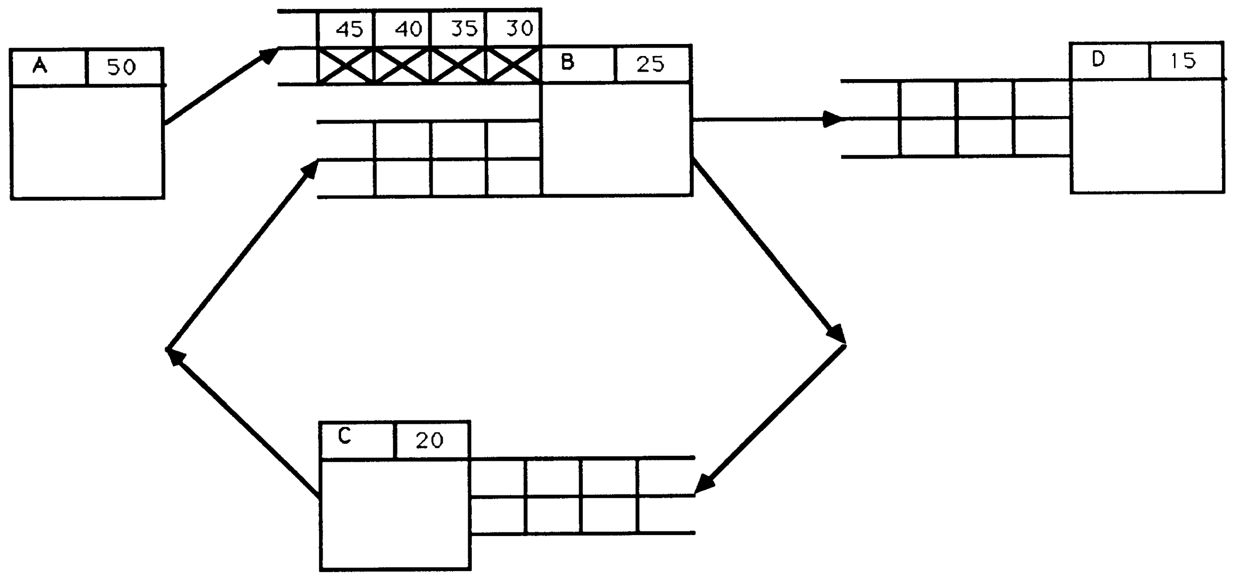


Figure 1.5 Local Deadlock

In most interesting simulations, the network contains a directed cycle, and if this is the case, then the conservative mechanism is vulnerable to deadlock. A momentary pause in the message stream along one or more arcs of the cycle may cause a local deadlock by *deadly embrace.* We define a *deadlocked* process as an *unterminated process permanently ineligible to execute.* Figure **1.5** provides an example where tasks **B**, **C**, and **D** are permanently unsafe. In this example, using a purely conservative mechanism leads immediately to deadlock since **B** can never process even the first message from **A**. This is a deadlock caused by the conservative mechanism, not by the simulation; therefore, the same simulation executed under the sequential event list method would complete successfully.

These two problems, memory overflow and deadlock, taken together contribute to the possibility of failure in large simulations. The chances are slim that a large complex simulation will succeed in terminating since these two problems may arise at any node and around any network cycle.

Even if there were a way to avoid these problems, the conservative mechanism does not extract enough of the available concurrency. An example of this excessive conservatism is shown in Figure **1.3**, where we see that **D** must block until B sends another message, because **B** is able to send a message whose time-stamp is less than 30. If **B** does that, then **D**'s waiting was necessary. However, if **B**'s message arrives with a time-stamp of, say, 50, then **D**'s waiting was in vain. In this case, node **D** could have safely processed its messages from **C**, resulting in increased concurrency as well as less risk of deadlock or memory overflow. This early processing of the messages from **C** reduces the chance of memory overflow because **D** no longer needs to keep copies of the messages from **C** after they have been processed. The early processing also reduces the chance of deadlock if there is a directed cycle **D-X- ... -X-B**, by preventing the flow of messages out of **D** from stopping. We believe that under a conservative mechanism, blocking *in vain* would happen a large fraction of the time. One solution would be for **B** to send null messages to **D**, just for synchronization purposes. On the other hand, increasing the number of messages is very likely to increase the time delay for all messages passing through the network. (This is a well known fact in routing theory, but may not be applicable to this system because we are dealing with a virtual network, not a physical one). Hence several arguments suggest that a purely conservative mechanism is not the best for concurrent simulation. Therefore, the following subsubsections will discuss how Peacock, Wong, and Manning; and Chandy and Misra try to alleviate the problems of a purely conservative mechanism.

1.2.2.1 The Link Time Algorithm

Peacock, Wong, and Manning [16] describe their Link Time Algorithm as a mechanism in which the sending node for each arc (link) maintains a *link time* for that arc. A link time is a lower bound on the time stamp of the next event message to be sent along the arc and is periodically communicated to the receiving node, which can use the information to determine whether it is safe to execute even if the input queue for the arc is empty of event messages. This seems to be equivalent to the following mechanism described by Chandy and Misra [5]. Each node of the simulation automatically sends extra time-stamped *null* messages along some or all of its output arcs whenever the local simulation time at that node changes. The null messages have no semantic content in the simulation and can be treated as events requiring no action. Like real transactions they play a part in the safe/unsafe decision, and processing them does cause the local simulation time of the receiving object to advance. This algorithm makes receiving nodes *safe* a greater percentage of the time. Hence, there is less waiting at each node, resulting in greater concurrency.

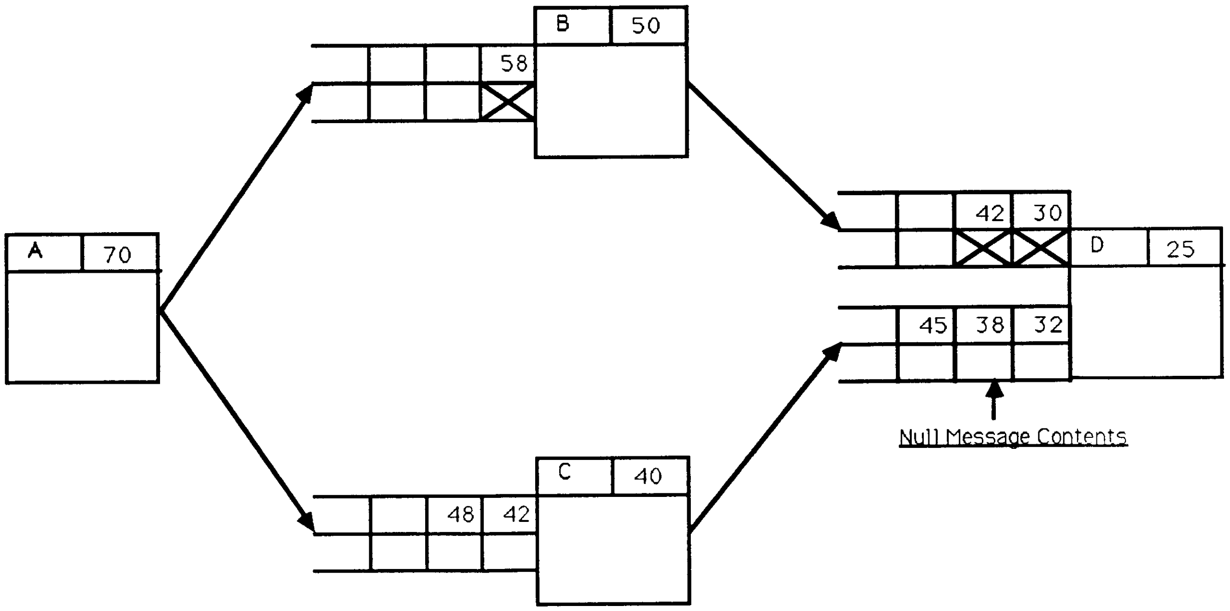


Figure 1.6 The Link Time Algorithm using null messages

Figure **1.6** illustrates this technique. In this diagram **D** has received three null messages from **C** with time stamps 32, 38 and 45. **D** can therefore process both of its messages in the other queue with time stamp less than 45 (30 and 42). Without this mechanism **D** would need to receive a message from **C** before it could process the appropriate messages.

The problem with using this approach is the increased number of messages in the system. It could therefore be possible for the system to be processing more overhead messages than real messages. This would therefore increase the overall delay of the system.

This portion of the Time Link algorithm solves the blockage issue. The second portion of this algorithm addresses the deadlock issue. This part of the algorithm requires that there be a non-zero minimum service time for all network nodes. This implies for each node, there is some number ϵ such that it cannot send a message with a time stamp less than $t + \epsilon$ in response to a message with time stamp t.

This requirement does avoid deadlock, but it constitutes a serious artificial constraint on the types of models that can be simulated. Many of the most frequently used service time distributions (exponential, Erlang, lognormal, etc.) do not have a nonzero lower bound. Queueing models using such distributions for service times either cannot be simulated or can be simulated only with distortion.

Another problem with the algorithm is that it may result in a *near deadlock* situation. To illustrate this point, consider Figures **1.7** -**1.9**. Suppose for this simulation that the minimum service time for a node is 0.001 and that the mean service time for a node is 1. In Figure **1.7**, node **B** wants to process a message from **A** with time stamp 37, but is being blocked by the empty queue from **C**. **B** has, however, sent null messages to **C** and **D**. In Figure **1.8**, the null messages have been received, and C has sent a null message to **B** with time stamp 35.002. However, **B** still cannot process the *real* message from **A** with time stamp 37. In Figure **1.9**, **B** again sends null messages. Continuing like this, messages with time stamps between 35 and 37 in increments of

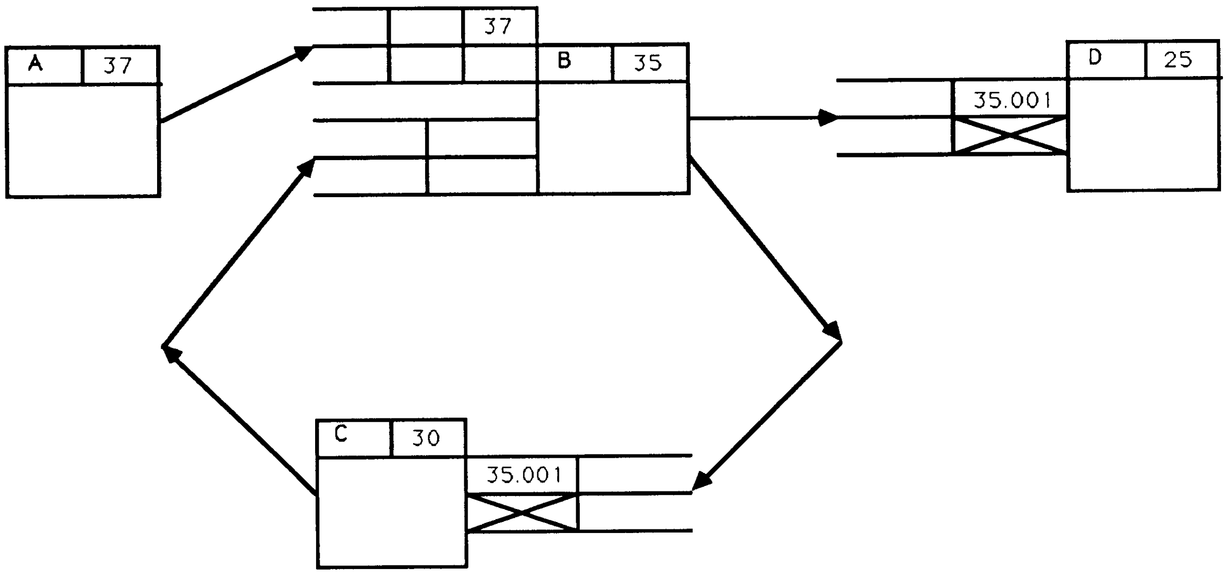


Figure 1.7 Using Minimum Service Times to avoid Deadlock I

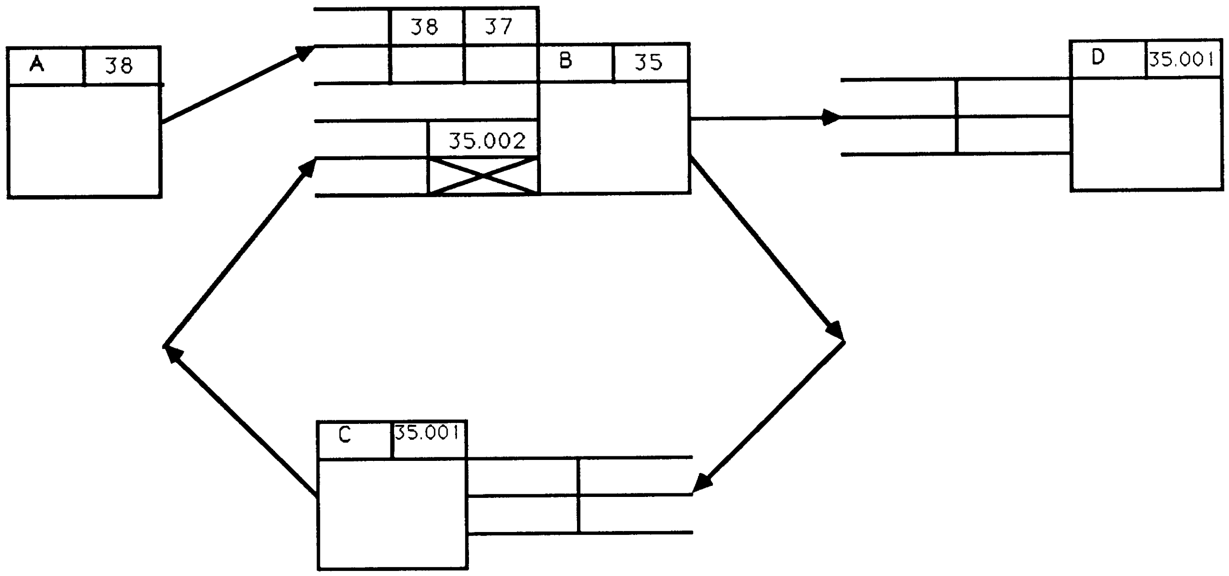


Figure 1.8 Using Minimum Service Times to avoid Deadlock II

0.001 will flow in the cycle **B-C-B** until **B** is able to receive the *real* message from **A**. This amounts to 2000 null messages sent and received with no concurrency. Although increasing the minimum service time reduces the number of messages sent, it cannot be set to too large a number or the simulation will become unrealistic.

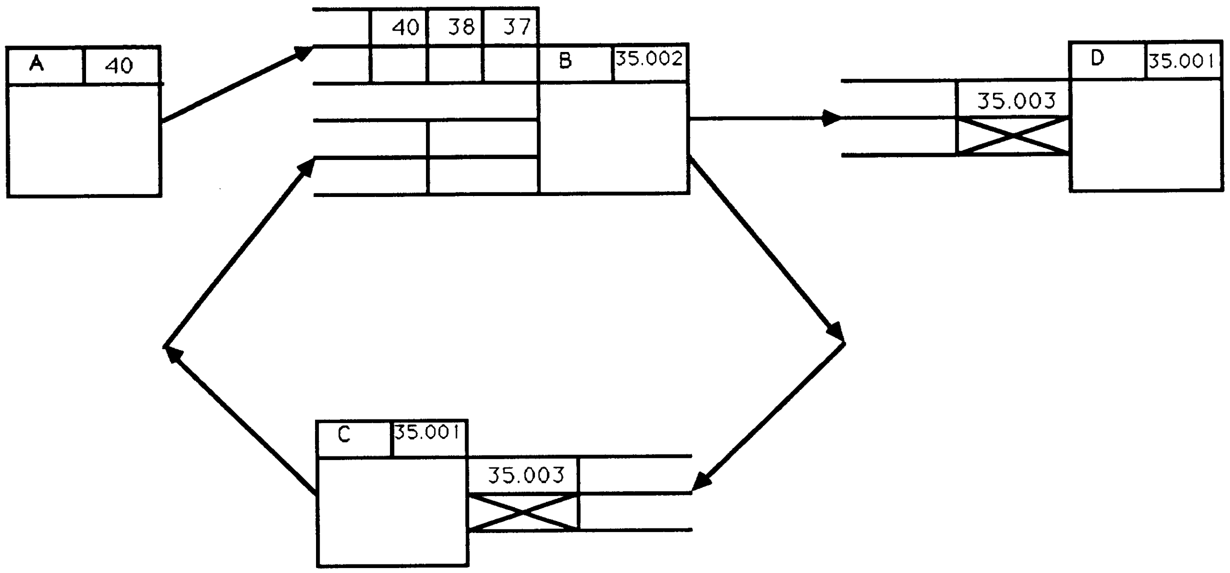


Figure 1.9 Using Minimum Service Times to avoid Deadlock III

1.2.2.2 The Blocking Table Algorithm

One way to avoid deadlock and possibly increase the concurrency over that available from the Link Time algorithm (at least theoretically) is to use the Blocking Table Algorithm (Peacock, Wong, and Manning [17]), which continually and incrementally computes part of a communication graph formed by the empty arcs (arcs terminating in an empty queue). With this information, a given node knows not only what nodes are directly blocking it, but also which nodes are blocking its blockers. A given node sends a request to each of its blockers for a lower bound on the time stamp of the next message sent. This information is maintained at each node in a blocking table. (see Peacock, Wong, and Manning [17] for details) This table allows a node to determine if a message can be processed based on global as well as local information.

According to [17], computing the actual blocking table - even incrementally - can be prohibitively expensive for large simulations. This is because a single event can both delete and insert new empty arcs into the subgraph of empty arcs. Therefore, they use an approximation to the graph calculation.

Our judgment of the Blocking Table algorithm is that in most cases the number of messages exchanged while computing the blocking table for each blocked node could exceed the number of messages exchanged during single processor event simulation. The system might spend more time deciding who goes next in the simulation than it would actually simulating. Since [17] provides neither details of the approximation nor experimental results, no conclusion of the algorithm's efficiency can be made.

1.2.2.3 Chandy and Misra Method

Chandy and Misra [6] consider a set of optimizations of the Network Paradigm, that addresses the problems of deadlock and memory overflow.

To address the problem of memory overflow, they require queues to have a finite length. In this case, a node blocks not only when it has an empty input arc, but also when the queue of the next destination is full. This mechanism clearly solves the memory overflow problem. Furthermore, it insures that the amount of storage used by a concurrent simulation will be within a factor of that used by a serial simulation.

However, this solution aggravates the deadlock problem because a node can block when either sending or receiving a message. Therefore, *deadly embraces* also can occur in cycles without regard to the direction of the arcs. This greatly increases the number of possibilities for deadlock. These local instances of deadlock can cause blockage at all nodes in the network because eventually the nodes will either have an empty input arc, or will fill the input queue of a blocked node.

To overcome deadlock, Chandy and Misra suggest that a deadlock-detection mechanism such as the Dijkstra's algorithm be used in tandem with the simulation. When deadlock is detected, the simulation stops and a sequential deadlock-breaking mechanism is employed.

Chandy and Misra do not expect the detection-breaking mechanism to be a bottleneck because they claim that it is fast and rarely needed. This may not be true because deadlock is more likely with finite queues. Furthermore, it appears that the mechanisms for breaking and detecting deadlock and restarting some of the processors must either be slow, performing a global analysis to detect the maximum number of processors that need to restart due to deadlock, or must use a faster analysis that detects fewer processors, but leaves the system in a near-deadlocked state.

1.2.3 Time Warp Mechanism

Time Warp is an asynchronous method that speeds up simulation by exploiting the concurrency that results from the programmer's decomposition of a model into interacting tasks. Time Warp was introduced by Jefferson and Sowizral [8,9,10].

The Time Warp method can be applied to a larger class of simulations than the Network Paradigm. There is no need to assume a fixed network topology connecting the simulation tasks; rather, each task may interact with any other at any time. Each task has only one input message queue; incoming messages are enqueued by sorting them into time-stamp order. In the Time Warp mechanism these time-stamps are called virtual receive times (VRT). The VRT of a message is the virtual time at which the task should process the message. The lowest VRT of any message on the task's input message queue is the LVRT of the task. There is no need to assume that messages are sent in increasing order along each arc. For example, we permit task **A** at simulation time 50 to send a message with VRT 100 to task **B**, and then later, at virtual time 70 to send a message with VRT 80 to the same task **B**. We maintain the assumption that the VRT on a message must be greater than or equal to the virtual time of the sending task at the moment of sending. This ensures that tasks cannot schedule events in the past.

Like the Network Paradigm mechanisms, the Time Warp is asynchronous, meaning that there is no global variable that represents the simulation clock; instead, each individual task maintains its own local Virtual Time (**LVT**). In any particular snapshot of a simulation some tasks will have **LVT** values greater than others, because the Time Warp mechanism is asynchronous.

The Time Warp mechanism always acts upon the messages in its input queue in VRT order until it exhausts the queue. In the Time Warp mechanism, snapshots of the task are taken from time to time. A snapshot is simply the state of the task at a particular virtual time. In contrast with the Network Paradigm, a Time Warp task never waits until it can *safely* process the next message (unless its input queue has been exhausted). A task can run any time there is at least one message on its input queue.

This aggressive approach risks the possibility that a message will arrive at a task whose **LVT** is greater than the VRT of the incoming message. In other words, it is possible for a message to arrive in the past. This message is called a straggler. The VRT of the straggler is called the preempting virtual time. The virtual time of the task immediately before the straggler appeared is called the previous virtual time. Straggler messages can cause other messages to be processed out of order–a serious problem, as each message may cause both a state change in the receiving task and the sending of additional messages. If even one message were to be processed out of VRT order, the results of the entire simulation might be invalid.

Since it is almost certain that there will be some stragglers in the course of a simulation, the Time Warp system provides a mechanism to ensure the simulation's integrity. Whenever a straggler arrives at a task's input queue, the Time Warp mechanism automatically restores that task to a state from a virtual time earlier than the preempting virtual time of the straggler, cancels any side effects that it may have caused in other tasks from the time of the preempting virtual time to the previous virtual time (possibly by rolling them back), and starts simulating all affected tasks forward again. In rolling back, messages that were sent between the preempting virtual time and the previous virtual time are canceled by sending anti-messages. Anti-messages are identical to the original message except for setting an additional bit in the message to indicate that the new message is an anti-message. While reprocessing the task, the processor will not send new messages until it reaches the preempting virtual time since these messages would have already been sent out correctly. Once the preempting virtual time is reached, processing continues normally and messages are sent until the previous virtual time. This set of actions is called a backup. A rolled back task may reprocess some messages that it processed before, but this time the straggler will be processed in its correct sequential position.

Even though the idea of rolling back to an earlier state may seem hopelessly expensive and clumsy, the mechanism is in fact quite economical. It rolls back only the tasks that must be rolled back, it rolls them back only as far as necessary, and it rolls them back at the first moment the information mandating the rollback becomes available. If rolling back can be done quickly enough and infrequently, then in large simulations the increased concurrency achieved will compensate for the overhead of the rollback mechanism.

The Time Warp mechanism is a game of chance. The object of the game is to minimize losses: real delay involved in wasted computation and rolling back. Every time a task processes a message **M** with VRT **t**, it takes a bet that no message with a VRT earlier than **t** will later arrive. If it wins that bet, no time is lost. If it loses the bet, some time is lost since there is a delay involved in restoring the task to an earlier state and then running it forward to the point when message **M** can be processed. The success of the Time Warp system is based on the assumption that most simulation programs are well-behaved and that stragglers will arrive rarely enough in the long run to make the gamble worthwhile.

A similar game is played in memory management in paging systems. Every time a running program makes a memory reference, it takes a gamble that the page referenced will be resident in memory. When the gamble is won, no time is lost, and the memory reference proceeds without delay; on the other hand, if the gamble is lost, then some time is lost, and the cost is the delay needed for disk accesses and page table manipulation before the memory reference can proceed. The success of paging systems is based on the empirical fact that programs are reasonably well-behaved. If simulations are well-behaved in message passing, then we can win our game. In addition, if message passing reaches a steady state, we will be able to minimize backups by analyzing statistics.[13] (Major portions of the material in Section **1.2** were taken from Jefferson [8]).

There are many issues that are not addressed in the Time Warp design by Jefferson. How are the processors mapped to tasks (scheduling)? How do users specify tasks and their initial input messages? How are these tasks initialized?

Although scheduling is not addressed in Jefferson's Time Warp design it can greatly affect the performance of the simulation system. The simplest way to implement scheduling is through a global task queue which is a sorted list of the tasks in LVRT order. Then, processors always process the queued task with the lowest LVRT. Since many processors may wish to access the task queue at any one time, the task queue must be locked every time a processor accesses it. As the number of processors increases, the task queue can become a major bottleneck because processors must wait a significant amount of time to access the task queue. Therefore, we investigated grouping tasks into partitions and distributing the task queue among the partitions. In this manner, we were able to lower the contention for the task queue and therefore lower the processors' waiting time.

Partitioning is another aspect, not addressed in Jefferson's design that can greatly affect the performance of a simulation. A partitioning method can effectively set priorities among the tasks to be executed. If done wisely, setting priorities can encourage task execution patterns that avoid backup. Therefore, a good partitioning method in conjunction with a good scheduling policy can reduce the amount of processing time and the amount of backup.

It is not always obvious how best to partition a simulation, but we found some good heuristics. One heuristic is to minimize the amount of inter-partition messages.

Once partitioning is used, major scheduling policy decisions must be made. The easiest and most straightforward scheduling strategy is the static partitioning policy. The static partitioning policy assigns a fixed number of processors to each partition. If there are no tasks to be executed in a partition, then a processor in that partition suspends itself until work arrives. We found that this type of policy did not do well when partitions had a large variance in their work loads over the time span of a simulation.

Another scheduling strategy is dynamic repartitioning. In dynamic repartitioning, each processor is still assigned a given partition. However, when there is no work for a processor in its given partition, then that processor relocates to another partition. This policy usually worked the best because processors remained in their partitions until it was absolutely necessary to relocate. Assuming we used a good partitioning method, it is advantageous to try to maintain the partition boundaries.

A final scheduling strategy is the continuous dynamic repartitioning strategy. In continuous dynamic repartitioning, a processor is not dedicated to a given partition. Instead, after processing each task, it selects its next task from a partition that is chosen by some algorithm (not necessarily the same partition as its last task came from). At first glance this strategy would seem to take advantage of the decentralization of task queues induced by partitioning, while still processing the the task with the lowest LVRT as in the unpartitioned case. When we used a good partitioning method, this strategy was not advantageous because it did not respect partition boundaries and the implicit priorities they induce.

In the Time Warp design, when a task backs up, all messages sent by the task between the virtual times from which and to which it is backing up are canceled. We call this the aggressive message cancellation policy. Many such messages may in fact be sent again after a backup, with exactly the same contents and virtual receive times, suggesting a new lazy message cancellation method where a message is not canceled until it is known that it will not be sent again by the backed-up task. We found that aggressive message cancellation worked well in simulations where each task's output typically depended on the task's state (local variables of the task) – since a backup would typically change the state, it was unlikely that many of the same messages would be re-sent after a backup. Conversely, lazy message cancellation worked well when the output was not typically state dependent. We shall refer to these message cancellation policies as synchronization recovery methods.

Therefore, the simulation system's performance can be greatly influenced by the following three design aspects that we have just discussed:

1. Scheduling
2. Partitioning
3. Synchronization Recovery Methods

These three design aspects are the most important because they are user controlled. In this thesis we shall try to understand how the user can use these controls to enhance the performance of the the simulation.

1.3. Approach

The first goal of this thesis is to implement an efficient message based system for concurrent simulation. The system uses many of the concepts developed for the Time Warp Mechanism, along with newly developed performance heuristics. Furthermore, the thesis also evaluates the effects that different parameters of the system have on simulation performance.

To optimize our system, we group tasks into partitions. One parameter that the thesis evaluates is the effect of different methods of choosing these partitions on the performance of the system.

Once partitions are chosen, we then need to choose a scheduling policy. That is, we need to assign priorities to the different tasks awaiting execution. There are three categories of scheduling policies: static partitioning, dynamic repartitioning, and continuous dynamic repartitioning. Our major efforts are focused on studying dynamic repartitioning. We develop several varieties of dynamic repartitioning and then compare their effects on simulation performance.

Another comparison that this thesis makes is between synchronization recovery methods. In the original Time Warp strategy, aggressive message cancellation is used. We also explore the lazy message cancellation policy. These two methods are compared over a wide spectrum of experiments.

Along with these comparisons, we also study the effects on the performance of the simulation of message queue lengths and of virtual-time and real-time delays. We investigate the effects of altering the average input message queue length of each task. The lengths of message queues can be decreased by increasing real-time delays and by introducing feedback. We also analyze the effects on the dynamics of simulations of changing the virtual-time delays between successive messages. Virtual time delays can be changed by modifying mean interarrival times (MITs). Real time delays can be modified by increasing the real time between successive input messages to our system or introducing feedback. Varying these parameters allows us to validate our theories about Time Warp. These theories, in turn, may suggest ways such as changing the partitioning or using a different scheduling policy to make a given simulation go faster and reduce backups.

1.4. Chapter Outlines

Chapter 2 presents the overall design of our system, introducing the concepts of simulation independent and simulation dependent subsystems. The general object oriented model used to build the overall simulation system is described. System functions and the system's user specifications are also described.

Chapter 3 introduces the simulation independent portion, modeled after Jefferson's Time Warp method[12]. Also discussed are modules that vary deterministic scheduling policies and interface with the simulation dependent portion. Finally, message cancellation policies are discussed.

Chapter 4 discusses all the different scheduling policies used: non-partitioning, partitioning, variants of dynamic repartitioning, and continuous dynamic repartitioning. Varying strategies processors can use to pick a new partition are discussed: fixed-list, circular-list, longest task queue and lowest LVRT (lowest virtual receive time). A final variation includes the estimation of virtual receive time, an idea built upon lowest LVRT dynamic repartitioning.

Chapter 5 covers various simulation dependent systems. These systems, which were implemented, include simulations of a queueing network, a digital circuit, and a butterfly network. Some of the modules used in one simulation were re-used in another, demonstrating their flexibility as building blocks. Several different partitioning methods were used on the butterfly network simulation; these are described.

Chapter 6 discusses extreme experiments; that is, best and worst case experiments for each of the scheduling policies. Each dynamic repartitioning strategy is distinguished with an example where it outperforms the other strategies, establishing that no scheme outperforms all others in all cases. It is thus valuable to study the different schemes in a practical setting to determine the circumstances under which each scheme performs the best.

Chapter 7 first describes a model of our system. The goal of the model is to understand how our system works and how we would expect it to behave under various sets of circumstances. Using this model we develop theories to explain the effects of scheduling schemes, partitioning methods, message cancellation strategies, message queue lengths, and real and virtual time delays on the performance of the simulation. Once the theories are presented, results of the actual simulations are given, comparing scheduling schemes, partitioning methods, message cancellation strategies, message queue lengths, and real and virtual time delays. The results of these experiments are used to confirm the theories presented with the model. Finally, we explain the applicability of these results to other simulation experiments.

Chapter 8 is the conclusion.

II. OVERALL DESIGN OF OUR SYSTEM

2.1. Introduction

In our system there are two different sub-systems, the simulation independent and the simulation dependent portions. The simulation dependent system consists of modules which are specific to a particular simulation. Three points are discussed: use of an object oriented programming approach with message passing to implement system design, six user system functions with which the user can compose the simulation dependent portion, and a list of items a user would need to specify while initializing our system.

2.2. Object Oriented Programming

The object oriented programming style used in our system is exemplified in Figure **2.1**. Here the procedure **make-closure** is defined as a function that returns another procedure. We call **make-closure** a procedure class. Inside the procedure **make-closure** an environment is defined as a set of bindings of additional parameters based on input arguments given to **make-closure**. Within this environment the sub-procedures foo_1 through foo_N are defined. These sub-procedures would differ if they were defined at top level because these functions when defined within **make-closure** generally depend on the values of $var_1 \ldots var_N$, as well as the arguments given to **make-closure**. Furthermore, if one of the functions inside **make-closure** perform side effects on one or more of the variables $var_1 \ldots var_N$, then only the environment inside of **make-closure** will be affected. However, if the functions were defined outside of **make-closure**, then the side effects would alter the global environment.

The application of **make-closure** to its arguments is shown in Figure **2.2** . Its result, *bar*, is called an instance or closure of the procedure class **make-closure**. The

```
(define (make-closure tt-table name outname ... arg )
 (let ((var1 (fun1  tt-table))
      (var2 (fun2 name ))
      (var3 (fun3 outname ))
                .
                .
                .
      (varM (funM  arg ))

    (define ( foo1 a11 a12 ... a1O )
      ... )

    (define (foo2  a21 a22 ... a2P )

      ... )
                .
                .
                .
    (define ( fooQ aQ1 aQ2 ... aQR  )
      ... )

    (define ( dispatch op )
      (cond ((eq op  'name1 )foo1 )
            ((eq op  'name2 )foo2 )
                  .
                  .
                  .
            ((eq op  'nameQ )fooQ )))

    dispatch))
```

Figure 2.1 Procedure Class Make-Closure

```
(setq bar (make-closure  table self them ... param))
```

Figure 2.2 Bar – An Instance of the Procedure Class Make-Closure

application of *bar* to its arguments is shown in Figure **2.3** . This style is called message passing and *bar*'s argument is called a message.

```
((bar 'name1) param1 param2 ... paramS)
```

Figure 2.3 Application of Bar Exemplifies Message-Passing

2.3. System Structure

Abstract task objects have two closure objects, one in the simulation independent and the other in the simulation dependent system. These closure objects are called images. The image of a task in the simulation independent system is always an instance of one of our fixed set of procedure classes. However, the image of a task in the simulation dependent system can be an instance of a wide range of procedure classes that depends on the simulation.

Messages are passed between and within the simulation independent and simulation dependent systems; however, a Time Warp message between abstract task objects is done in the special manner illustrated in Figure **2.4** . In this example, task A is sending a Time Warp message to task B. The images of A and B in the simulation dependent system are A_1 and B_1 respectively. Similarly the images in the simulation independent system are A_2 and B_2. The first step in this process is the sending of $message_1$, whose contents is the Time Warp message, from A_1 to A_2. The next step is the sending of $message_2$ from A_2 to B_2. Next B_2 sends $message_3$ to the message queue of task B. Subsequently, $Message_4$ will be sent from the message queue to B_1 at the appropriate virtual time specified by the Time Warp message.

Although we have discussed the communication between task objects, we have not discussed how the destinations of object oriented messages are obtained: that is, we have not shown how objects A_2, B_2, and B_1 are obtained. A naming strategy allows us to obtain these objects.

Every abstract task in our system is assigned a name. This permits tasks to be identified at run time. Tasks are created dynamically at run time and hence cannot be referenced a priori.

The task name to task object table (*tt-table*) is a map of task names to their images

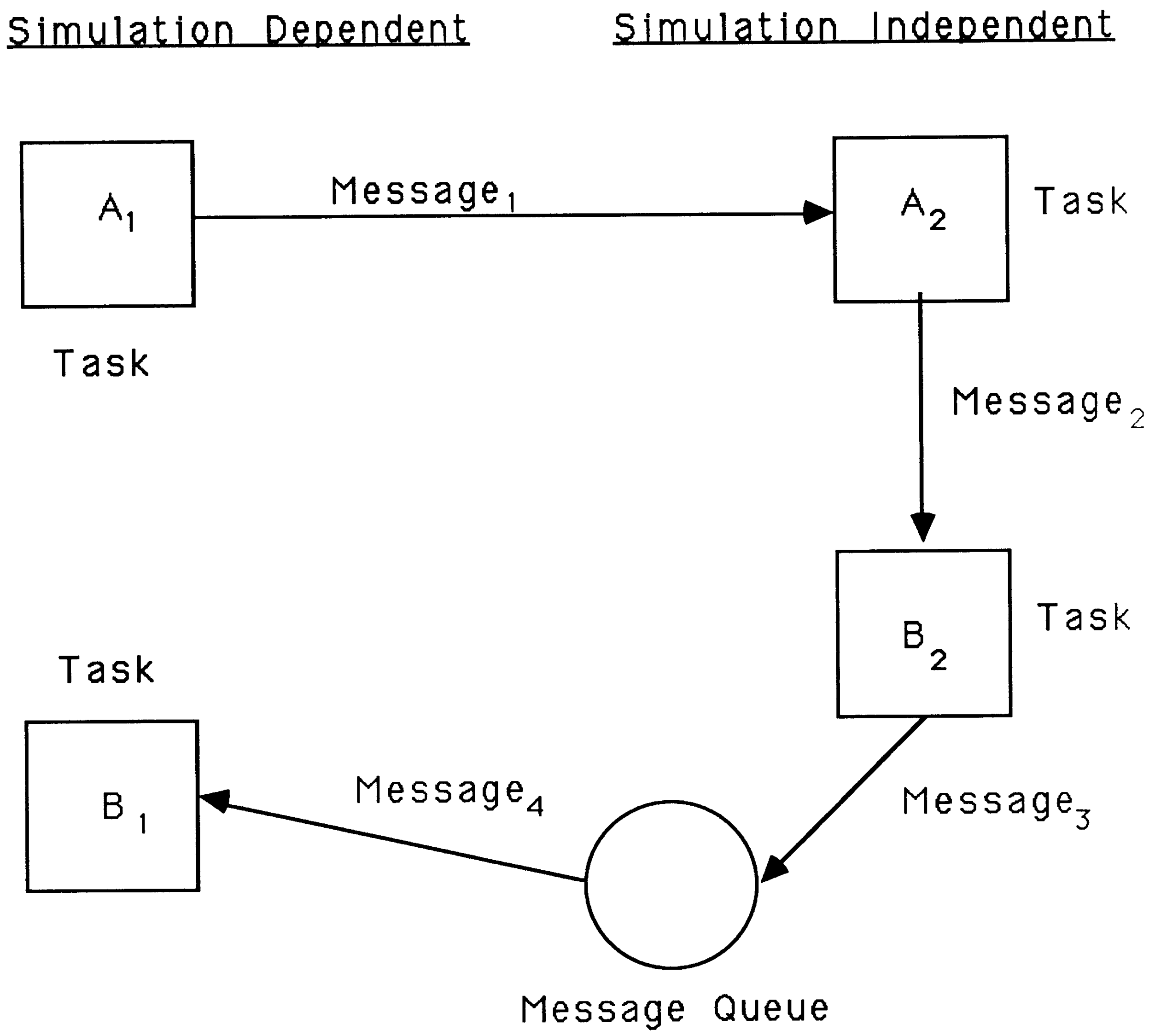


Figure 2.4 Abstract Task Object A sends a Message to B

in the simulation independent system. An entry is added for a given task when the task is created. This entry allows tasks in the simulation dependent system to access images in the simulation independent system.

Tasks in the simulation independent system obtain the images of tasks in the simulation dependent system differently. Instead of using *tt-table*, a task in the simulation independent system maintains a pointer to its counterpart simulation dependent image. This is more efficient because tasks in the simulation independent system need only access their counterpart images and no others. Tasks in the simulation dependent system, however, may need access to several tasks in the simulation independent system in order to send messages to them; therefore, a table more compactly represents this information, rather than having pointers from each image in the simulation dependent system to many different simulation independent images.

We now return to Figure **2.4** . When task A wants to send a Time Warp message to task B, the Time Warp message is initiated by A_1. A_1 knows only the names of the tasks A and B, and recovers A_2 and B_2 from *tt-table* with this knowledge of the tasks' names. B_2 recovers B_1 via its pointer to B_1. Hence the path A_1 A_2 B_2 B_1 can be referenced.

2.4. Environment

Every task has an associated set of state variables. In order to perform backups, previous task states must be obtainable. This set of previous task state variables is called an environment, but this environment of a task should not be confused with the environment of a closure.

The environment of a task can be obtained by both of the task's images. It is pointed to by the image of the task in the simulation independent system. The simulation dependent image first obtains the simulation independent task by a *tt-table* lookup, and then references the task's environment via its counterpart's pointer.

2.5. Simulation-Dependent System Functions

A set of functions need to be defined in order to write simulation dependent procedure classes. These functions include:

1. Retrieving a task closure object (image) in the simulation independent system corresponding to a task name in the simulation dependent system.
2. Retrieving the task name in the simulation dependent system from the task closure object in the simulation independent system.

3. Retrieving a task's environment closure object in the simulation independent system.
4. Storing values of variables in the current task's environment.
5. Retrieving values of variables from the current task's environment.
6. Sending messages from one task to another via their images in the simulation independent system.

((tt-table **'id-name)** *symbol-name):* [function]

This function returns the image of the task *symbol-name* in the simulation independent system, which is a closure object representing the task. The variable *symbol-name* is the symbol-name of the image of the task (for example A in Figure **2.4** is the symbol-name of a task and A_1 or A_2 are the task's images.)

((tt-table **'name-id)** *task-closure):* [function]

This function is the inverse of the above and is used to retrieve the task's symbol name. The variable *task-closure* is the closure object representing the task's image in the simulation independent system.

((task-closure **'get-task-env** *)):* [function]

This function returns the environment closure object for the task associated with the task closure object *task-closure.* The environment closure object is used to store and retrieve variables in the environment of the task.

((env-closure **'set-env-var)***name value):* [function]

This function saves the *value* of the variable having symbol name *name*, in the environment represented by the closure object *env-closure.*

((env-closure **'get-env-var)***name):* [function]

The parameter *name* is the symbol name of a variable in the environment represented by the closure object *env-closure.* This function returns the value of the variable, and is used to retrieve the values of variables stored in the environment of a task.

((sclosure **'send-mess)***rtime stime data sclosure rclosure anti):* [function]

This function sends a message from the source task (stask) to a receive task (rtask). The image of stask in the simulation independent system is *sclosure* (similarly *rclosure* is the image of rtask). The variable *rtime* is the VRT of the message. It is the time when rtask should processes the message. The variable *stime* is the current virtual time of stask. The variable *anti* is used to send anti-messages during backups, and is set to zero for user-specified messages.

The following example (Figure **3.5**) illustrates how to use system functions to define a procedure class. The procedure class in our example represents the definition of a wire in a circuit simulation. The application of the procedure **make-wire** creates a wire. The procedure class initializes a wire by attaching it to appropriate circuit components and attaching an initial value to it (line 14). Along with these initializations, the application of the procedure class returns a closure that represents a wire. When this wire is applied to a message, the message is then propagated to all circuit components to which the wire is attached.

The application of **make-wire** to its arguments occurs at initialization. However, before this occurs, the user specifies the variables *circuit* and *signal-value.* The variable *circuit* holds holds all of the tasknames of the circuit elements to which we want to attach the wire. The variable *signal-value* represents the value of the wire's signal value and is initialized to zero. These are placed in the task environment object at initialization. This is not part of **make-wire** so that a single wire can be connected to different sets of circuit components. Then the procedure class **make-wire** is applied to its arguments. The arguments in this case are the *tt-table* and *name.* The *name* variable is the symbol name of the task representing the wire in the simulation dependent system. *tt-table* is the closure object performing the table mapping function. The local environment will be the first thing defined in the procedural definition. Within this environment all of the system functions are initially referenced and bound to local variables as illustrated in lines 2-7. This use of local variables will allow access to the system functions without having to look them up again.

```
[1] (define (make-wire tt-table name)
[2]   (let* ((in (tt-table 'id-name))             ; Mapper from names to Simulation Indep. images
[3]          (wclose (in name))                   ; Simulation independent image of name
[4]          (send-mess (wclose 'send-mess))      ; Proc that sends a message from a source to dest task
[5]          (env-close ((wclose 'get-task-env))) ; Environment closure object associated with name
[6]          (get-var (env-close 'get-env-var))   ; Function that retrieves a variable value in env
[7]          (set-var (env-close 'set-env-var)))  ; Function that binds a name, value pair in env
[8]     (set-var 'circuits (mapcar in (get-var 'circuits)))
[9]     (define (call-each circuits time stime value)
[10]      (cond ((null circuits) nil)
[11]            (t (let ((cir (car circuits)))
[12]                 (send-mess time stime value wclose cir 0)
[13]                 (call-each (cdr circuits) time stime value)))))
[14]    (call-each (get-var 'circuits) 0 0 (get-var 'signal-value))
[15]    (define (reg-mess mess)
[16]      (let* ((time (mess-rtime mess))                 ; The time component of the message
[17]             (data (mess-data mess))                  ; The data component of the message
[18]             (signal-value (get-var 'signal-value)))  ; The wire's previous signal value
[19]        (cond ((<> signal-value data)
[20]               (set-var 'signal-value data)
[21]               (call-each (get-var 'circuits) time time data)))))
[22]    (define (dispatch mess)
[23]      (cond ((equal mess nil) nil)
[24]            (t (reg-mess mess))))
[25]    dispatch))
```

Figure 2.5 Example of a Procedure Class–Make-Wire

The next action to occur is that tasknames in the variable circuit are replaced with their respective images in the simulation independent system (line 8). Then, the sub-function **call-each** (lines 9-13) is defined. This procedure sends messages to each of the circuit elements connected to the wire, telling the elements that the wire is an input to them. This function has three other arguments; *time, stime,* and *value. time* represents the desired receive time of the message, *stime* represents the desired send time of the message, and *value* is the desired signal value of the input wire to the circuit elements. Then, the wire being created is initialized by its connection to all components of the circuit at time zero with a signal value of zero (line 14).

Then the sub-procedure **dispatch** is defined (lines 22-24). This procedure dispatches messages to the **reg-mess** procedure. That is, it propagates a given message onto the elements attached to the wire. The procedure **dispatch** is the value returned by the procedure class **make-wire**. Hence, a wire is a procedure that when given a message encoding a signal propagates the signal onto the circuit elements attached to the wire.

The sub-procedure **reg-mess** (lines 15-21) is then defined. The procedure propagates a given signal onto the devices attached to the wire. This procedure is a function of a message consisting of time and data components. The function first checks to see if the message changes the signal value of the wire. That is, it checks to see if the data component of the message differs from the value of the variable *signal-value*, which contains the previous signal value on the wire (lines 18-19). Then all of the circuit elements that are attached to the wire are notified of the change and the new signal value of the wire is saved in the environment.

2.6. Initialization

The following is a list of parameters that must be user specified:

1. **Groups.** Groups, or partitions, are collections of tasks grouped together for scheduling purposes, as discussed in Section **1.3**. Associated with each simulation is a group list, which is an unordered list of the groups. The user must specify how to apportion the different tasks into these groups.
2. **Initial Processor Distribution.** The initial processor distribution states how many processors will originate in each of these groups. The association of processors with groups may change depending on the scheduling mechanisms used.
3. **Initial Group List Order.** Initial group list order is the order in which each

group with its associated tasks gets initialized.

4. **Procedure Object.** Procedure classes must be specified for each different type of task in the simulation. They include the different instantiations' task names and the names of tasks with which each task communicates. Each of these procedure classes return a procedure capable of processing any type of message that it could receive from any task.
5. **Environment.** The environment of each task, that is, the set of state variables associated with each task, must be defined.

2.7. Summary

Using object oriented programming is nice because each different type of module in our system can be represented by a procedure class. In this manner the simulation system is completely specified by the different procedure classes. This allows us to represent the simulation independent system by a fixed set of procedure classes, while allowing the user to define the simulation dependent system by a variable set of procedure classes. One of the most useful representations is an abstract task object with an image in both the simulation independent and the simulation dependent systems. These images are simply instances of procedure classes. Representing a task in this manner allows differentiation between its simulation independent and simulation dependent parts. This saves the user from having to rewrite the simulation independent portions over again. Only a single set of user procedure classes need be used to represent any simulation system, thus making our system a very general form of discrete event-based concurrent simulation.

In the second half of this chapter we presented a set of six system functions. These system functions tell exactly what the simulation dependent system can ask the simulation independent system to do. Section **2.5** is a user's manual for these functions.

III. SIMULATION INDEPENDENT SYSTEM

3.1. Introduction

Within the simulation independent subsystem there are two types of modules. One set of modules implements Time Warp, while the second comprises the scheduling and interface portions. Many missing Time Warp details had to be designed and specified. This chapter begins with a brief description of our implementation of Time Warp followed by a description of the modules that are involved in scheduling and interface. Variants of the basic scheduling system are either built upon or are special cases of the dynamic repartitioning strategy. Finally there is a brief discussion concerning lazy message cancellation.

3.2. Time Warp Implementation

We have described the Time Warp system designed by Jefferson in Chapter 1. Here we discuss our implementation of its simulation independent portion by giving a brief description of each module that is required and how the modules interact.

Our simulation independent Time Warp system consists of the following different data objects and procedures:

1. Messages
2. Tasks
3. Forward Message List
4. Backward Message List
5. Time-Snapshots and Time-Snapshot Lists
6. Environments

Each task has a forward message list, a backward message list, a time snapshot list, and an environment. These message lists contain messages that the task has sent or received. Messages are used as inter-task synchronization mechanisms since messages can only be sent after the source task's LVT, and only processed at a virtual time after the destination task's LVT. This prevents any task from progressing far ahead of the other tasks with which it communicates.

The environment consists of variables which represent the state of the task, and which may change due to the arrival of messages from other tasks. The snapshot list contains a list of snapshots, that is, copies of the task's environment at particular times. These snapshots are vital in order to restore the state of the task should a backup occur.

3.2.1 Messages

Messages are the means by which tasks communicate and are also the means of global task synchronization, since without messages a network of tasks would be completely decoupled. Messages prevent tasks from becoming overextended in virtual time. In other words, tasks cannot progress too far ahead of other tasks, otherwise messages from tasks that are behind will force them to back up.

When a message is sent from one task to another a copy of the message is placed into the backward message list of the source task and into the forward message list of the destination task. Each message is a data object consisting of a virtual send time, a virtual receive time (VRT), the source task object, the destination task object, the data and the anti-message flag. The source task object is necessary, since the destination task may need to know where the message originated in order to process the message.

3.2.2 Tasks

A task is a closure object that is linked to a message processing closure in the simulation dependent system. Each execution of a task may be performed by a different processor. This execution is the processing of one message by that task. An execution of a task begins with a task reading a message on its forward message list. If the VRT of the message is greater than the local virtual time (LVT) of the task receiving it, then the processor updates the LVT of the task to the VRT of the message and applies the message processing closure to the contents of the message. This processing may result in the sending of subsidiary messages to other tasks.

If the VRT of the message is less than the LVT of the task, then a backup is executed. The VRT of the message that caused the backup (preempting message) is referred to as the preempting virtual time. Backup involves restoring the task to the most recently saved state (including its list of incoming messages not as yet processed) prior to preempting virtual time; then all messages sent between preempting virtual time and the LVT of the task prior to backup must be negated (aggressive cancellation). After this, the task is executed forward to the VRT of the preempting message without the sending out of additional messages. Then, once the message which caused the backup is handled, the task can go back to its normal routine of reading the next message on its input queue.

3.2.3 Forward Message List

The forward message list consists of message data objects received by the task which are sorted into virtual time order. This list is separated into two parts, the *processed message queue* and the *unprocessed message queue.* The *processed message queue* consists of messages which have already been processed and are then stored into decreasing VRT order. The *unprocessed message queue* consists of messages yet to be processed, stored in increasing VRT order. In this way the VRT of the first message of the *unprocessed message queue* is the LVRT of the task. When the processor executes this task, it will remove this first message from the *unprocessed message queue*, execute the task's message-processing closure on it, and then place the message into the first slot of the *processed message queue.*

If a message is received that has a VRT less than the task's LVT, then the task must back up, which is accomplished by transferring all of the messages with a VRT greater than the backup time from the *processed message queue* to the *unprocessed message queue.* After the backup the task proceeds as usual, reading the current message from the *unprocessed message queue.* Any anti-messages that are received either delete their corresponding messages from the *unprocessed message queue* or cause another backup and then delete the corresponding message from the *processed message queue.*

3.2.4 Backward Message List

A task's backward message list is a listing of all messages that have been sent by that task. Should a task back up, this list is then used to cancel the messages sent since the time of backup by sending anti-messages. Of course, these anti-messages may cause additional backups.

3.2.5 Time-Snapshots

Snapshots are a means by which the system can save the environment of a task at a specific time. Before processing a new message, a snapshot of the task is created which includes the current LVT of the task. For a particular virtual time, all of the current values of the task's state variables are saved in the snapshot, and then the snapshot is added onto a snapshot-list. Snapshots are used only in backups. If it is decided that a task has to back up to a certain preempting virtual time, then the snapshot with the largest virtual time less than the preempting virtual time is taken from the snapshot list. The task's environment is then restored to that snapshot state.

3.2.6 Environments

The environment data object contains two lists; a list of the names of all of the task's state variables and a list of their values. In this way all of the state variables of a particular task can be shielded from all of the other tasks, so that the system works in a modular fashion. By making a copy of the current task's environment data object and then tying it to the appropriate virtual time, we make a snapshot state of the task. Whenever a task has to be restored to a prior virtual time, the requisite snapshot object is searched out from the snapshot list.

3.3. Dynamic Repartitioning

In certain simulations a set of tasks may be highly coupled; in other words, each task depends upon messages sent by other tasks in the set. In this case the ability to continue processing one task in the set may be stalled by another task. It is then beneficial to permit only a few of these tasks to be running during any real time interval. In order to accomplish this, sets of tasks that are highly interdependent can be grouped together by the user. Some tasks may also be highly independent of each other, suggesting that they should process concurrently in separate partitions. In either case, the user should specify which tasks are dependent and which are independent by choosing the initial partitioning.

We investigated a strategy called dynamic repartitioning. In this strategy, there is an initial assignment of processors to partitions. A module called the *monitor* may subsequently reassign processors. A processor can only relocate to another partition after it finishes executing a task. The *monitor* attempts to determine more efficient dynamic groupings of processors during simulation execution. This dynamic load balancing may occur many times during a simulation.

The four basic modules that are used in the scheduling and interface between the simulation independent and simulation dependent portions of our system are:

1. Groups
2. Group Lists
3. Monitor
4. Task Queues

They are discussed below.

3.3.1 Groups

Each group data object represents a partition, and is associated with a collection of tasks and a task queue. Basically, groups appear as separate smaller simulations which have artificial fire walls between them preventing the tasks of one partition from relocating to another partition. Messages pass freely between partitions, but the tasks do not. Synchronization between partitions is achieved entirely by messages; however, inter-task synchronization within a partition is provided by both the task queue and messages.

3.3.2 Group List

The group list data object is simply an unordered list of group data objects. Depending on which dynamic repartitioning method is used, the group list may be modified. In the circular-list dynamic repartitioning, the group list is circulated every time a processor switches partitions. Circulation is accomplished by moving the first group to the end of the list.

3.3.3 Monitor

The *monitor* is a function that interfaces the simulation independent and simulation dependent portions of the simulation. One of the *monitor's* responsibilities is for the initial startup of the simulation. First the *monitor* makes a group list object and then places all of the groups in the list. It then makes a task queue object for each group and subsequently places all of the tasks associated with that group onto the task queue. Next the *tt-table* is created.

Once the creation of the objects is complete, the *monitor* initializes each task queue by creating each task's message-processing closure and then placing the closure into the task object which is the simulation independent image of the task. The closure is user specified and should be capable of handling any message sent to that task. This closure is an instantiation of a procedure class like the procedure class shown in Figure 3.5.

Next, the *monitor* creates the number of physical processors specified by the user for each partition. Each physical processor will execute the main loop of the *monitor* until the simulation is complete.

In this main loop each processor takes one group off the group list. Then the processor gets the corresponding task queue for the group that it chose. This task queue is sorted in terms of LVRT, so that the first task on the task queue is to be executed next. (A task's LVRT is the VRT of the first message on the forward message list of that task.) Should the task queue be empty, the processor will seek work at some other partition. If a processor decides to switch to a particular partition, then it changes its group and takes work from the new group's task queue. This new task queue corresponds to a new group; in essence, we relocate the processor to the new group. If the processor still finds no work outside its partition, then it will suspend itself inside its original partition.

If the task queue is not empty, then the processor grabs the next task on the queue and seeks its associated forward message list. Since the forward message list is sorted in terms of VRTs, the processor will try to process the unprocessed message with the lowest VRT on the forward message list. If the message's VRT is less than the corresponding task's LVT, then backup is required. Otherwise, the task's corresponding message-processing closure is applied to the message. Once this is complete, the task is placed back on the queue if there are more messages in its forward message list. If this list is empty, then the task is left off the queue pending the arrival of a new message. The processor will then return to the top of the main loop and proceed.

3.3.4 Task Queues

A task queue is a sorted list of executable tasks. There are three types of tasks: executable, non-executable, and executing. An executable task is one that has messages on its forward message list. A non-executable task has no messages on its forward message list. An executing task is one which is currently being executed by a processor. There are three major functions that the task queue object may perform on the tasks in its partition: adding a task to the task queue (*add-task*), updating the position of a task on the task queue (*update-task-queue*), and removing the next task for processing (*next-task*). Since many processors may wish to access the task queue at the same time instant, the task queue must be locked before any operation is performed on it.

Add-task first checks to see if there are any suspended processors which may immediately be given the task. Otherwise, the task is inserted into its place in the task queue which is arranged in increasing LVRT order. *Update-task-queue* simply changes the order of tasks in the task queue if their LVRT order has changed. For example, if a new incoming message arrives with a lower VRT than any messages on a task's forward message list, it changes the task's LVRT and thus requires the task to be moved to a different place in the task queue. *Next-task* grabs the next task off the queue for execution; if there are none, then the out-of-work processor relocates itself to another group, which is determined by the scheduling algorithm. If, after searching all of the other partitions, the processor still finds no work, then it will suspend itself.

In general, we wish to sort the task queue in such a way that the tasks which are farthest behind will get executed first. If we were to use LVT instead of LVRT to sort the task queue, then this would create a problem if a preempting message arrived at an idle task because the early VRT of the preempting message would not be reflected in the position of the task on the task queue. When this task finally gets backed up it may precipitate even more backups, all of which could have been avoided had the task been scheduled earlier. Unfortunately, this phenomenon can also occur with regular non-preempting messages. For an example consider $task_1$ with $(LVT_1, LVRT_1)$ and $task_2$ with $(LVT_2, LVRT_2)$; if $LVT_1 < LVT_2$ and $LVRT_1 > LVRT_2$, then the processing of $task_1$ first is wrong and could cause additional backups. The only real reason for keeping LVT is for knowing when to back up. If an executable task receives a new message with a $VRT < LVRT$ while using LVRT to sort the task queue, then the LVRT of the task is reset and the task is resorted onto the task queue. If an LVRT sort, rather than an LVT sort is used, a processor will always select for execution the task with the lowest LVRT within its partition.

3.4. Lazy versus Aggressive Message Cancellation

The original Time Warp mechanism used aggressive message cancellation during backups. When an task **X** was rolled back from simulation time 100 to 90, Time Warp immediately sent anti-messages for all messages that **X** had sent at or after time 90.

With the lazy cancellation method the rollback to time 90 does not immediately cause anti-messages to be sent. Instead, after the rollback, as task **X** simulates forward from time 90 to time 100, the set of new output messages **N** that **X** requests for transmission is compared to the set of old output messages **O**, whose transmission **X** requested the last time it simulated from 90 to 100. The messages in the set $\mathbf{O} - \mathbf{N}$ are canceled with anti-messages, because they never should have been sent; those in the set $\mathbf{N} \cap \mathbf{O}$ are ignored because they have been transmitted; while those in the set $\mathbf{N} - \mathbf{O}$ are transmitted to their destinations.

Lazy cancellation usually produces fewer anti-messages and secondary rollbacks than aggressive cancellation does. On the other hand, lazy cancellation delays the backup of other tasks and thus potentially increases the total amount of backup. The two cancellation strategies are compared in Chapter 7.

3.5. Summary

In this chapter all of the modules that make up the simulation independent system were presented. The crux of the chapter concerns the design of the new modules used to implement a wide variety of scheduling policies. Together with the *tt-table* module discussed in Chapter 2, these modules form the interface between the simulation independent and simulation dependent subsystems. They are necessary, but were not specified by any previous description of Time Warp. Finally, lazy message cancellation was introduced, in which messages are canceled only when they absolutely need to be.

IV. SCHEDULING

4.1. Introduction

Each of the different scheduling policies studied were either developed to increase the performance of our simulation system or to be used in baseline experiments. This chapter discusses the changes to existing modules that were required in order to implement those different scheduling policies and explains why those policies are interesting. In addition to deterministic scheduling schemes, this thesis also investigated an adaptive method. The question is whether the amount of effort to perform the adaptive method is too much of a price to pay for the added performance. The last scheduling scheme involves continuous dynamic repartitioning, in which processors are able to move to a new partition after the execution of a single task.

4.2. Non-Partitioning

In the non-partitioning case the simulation system acts as one large partition which continuously executes tasks with the lowest LVRT off the partition's task queue using physical processors. All other scheduling schemes are derived from this one. This policy can be implemented on our system by simply running the simulation with only one partition and suspending processor reallocation. Since there is only one partition, processor relocation does not make sense; therefore, processors that find no work in the partition simply suspend themselves until more work arrives. Non-partitioning will act as a base line on which to judge the merits of partitioning. A potential problem with this policy is task queue contention since many processors may need to access the lone task queue at any one time.

4.3. Partitioning

The partitioning case is very similar to the dynamic repartitioning strategy discussed in Section 3.3 except that processors do not relocate when there is an absence of work in their own partition. Rather than relocating, they are suspended. This policy can be implemented on our system by suspending processor reallocation. Static partitioning will become our base line on which to judge the merits of dynamic repartitioning. A potential problem with static partitioning is that if one partition had no work, then all of its processors would suspend, which would be wasteful if there is work in other partitions that needs to be done.

4.4. Dynamic Repartitioning

The following sections discuss the different algorithms performed on the group list to determine the partition to which a processor from an idle partition chooses to relocate. These different types of dynamic repartitioning strategies are:

1. Fixed Group List
2. Circular Group List
3. Group with Longest Task Queue
4. Group with Lowest LVRT
5. Estimated LVRT or EVRT

4.5. Fixed Group List

Fixed group list is the basic dynamic repartitioning strategy that was discussed in Chapter 3. When a partition's task queue becomes empty, all processors within that partition not executing a task seek work elsewhere. In seeking work, a processor scans the other partitions (i.e. groups) in the order in which they appear on the group list starting at the beginning of the group list. In scanning a partition, the processor retrieves the task queue for that partition. If this task queue is non-empty, then the processor relocates itself to that partition by moving its task scan to this new task queue. If the task queue is empty, then the processor scans the next group on the group list for work. If none of the other partitions have any work, then the processor suspends itself in its original partition. Because of the order in which the group list is scanned, partitions that are at the front of the list tend to receive idle processors first.

A potential problem with this method is that the order in which the processor scans partitions to find work is the same as the order of the groups on the group list. This tends to favor idle processors relocating to the first group on the group list, which is bad if groups at the end of the group list have the lower LVRT's. If processors are not assigned to the tasks that have the lowest LVRT's, then backup may occur. The advantage of the strategy is its simplicity. Very little overhead is required in order to determine where to relocate idle processors.

4.6. Circular Group List

Discussed briefly in Chapter 3 as an example of alternative dynamic scheduling policies, this scheme works exactly like the fixed group list scheme, except that after a processor relocates, the group list is circulated by moving the first group to the end of the list. Processors subsequently seeking work scan the revised group list. In this manner the scan is not biased toward any particular portion of the group list.

A partition has work to do if its task queue is non-empty, i.e. the partition has non-executing executable tasks. A potential problem is that this selection process does not distinguish between partitions that have much work to do from those that have little. Therefore a partition that had just one task to execute, but was at the front of the group list, could preempt a processor from relocating to a partition that had many tasks to be executed.

4.7. Group with Longest Task Queue

The longest task queue strategy is more complex. This policy tries to provide a fair relocation policy by relocating processors to the partition with the most tasks to be executed.

When a partition has no executable tasks, it is called an idle partition. Any processor within that partition which is not executing a task should relocate to a partition where work exists. The way this mechanism works is that a processor from an idle partition first retrieves the task queue of each group in the group list. Then the processor relocates itself to the partition with the longest task queue in essence, the partition with the most work. If all of the task queues have zero length, then the processor suspends itself in its original partition.

A potential problem with this method is that the partition chosen may not be the partition that is farthest behind in the LVRT sense. Hence, by executing the chosen partition, a backup may soon result. A partition with the greatest amount of work may still be far ahead (in terms of virtual time) of a partition with less tasks in its task queue.

4.8. Group with Lowest LVRT

The lowest LVRT strategy chooses the partition that is farthest behind in LVRT. In each group's task queue, the task at the front of the task queue has the lowest LVRT in the partition, which is also the LVRT of the partition. In the lowest LVRT strategy, a processor seeking work first retrieves the task queue of each group in the group list. Then the processor relocates itself to the partition with the lowest LVRT – in essence, the partition whose next message to be processed is farthest behind. If all of the partitions have no messages to be processed, then the processor suspends itself in its original partition. LVRT scheduling tries to give more processing power to partitions that are farther behind in virtual time.

A potential problem with this method is that it is susceptible should there be wide deviations in message arrival times. Examples of this are shown in Sections 6.4.2–6.4.4.

If a simulation has messages that arrive randomly based on a known probability distribution, then it may be possible to estimate the VRT of the next message for any particular task based upon VRT's of the task's previous messages. By determining the likelihood of backup if the first task on a partition's task queue is processed, one can decide whether to process the task. This may not be possible at the start of the simulation, but after the simulation has progressed for some time it may be possible. It is at this time that we say the simulation is in steady state.

4.9. Estimation

4.9.1 Motivation

After an initial startup period, a simulation may reach a steady state. From this state one may be able to approximately calculate a probability distribution on the virtual times of message arrivals using the statistics that have been gathered about message interarrival times. These probabilities can be used to estimate the task's next message virtual receive time (EVRT). If a task's EVRT is a good estimate, then it can be used to better sort the task into the task queue, thus increasing performance. The task queue should now be sorted in increasing order by the value of LEVRT=LVRT – EVRT, in order to discourage the processing of messages with LVRT >> EVRT. Assuming our model is correct, our estimate predicts that the next message that this task should process will have an VRT of approximately EVRT. If LVRT >> EVRT, our estimate indicates that we should hold up processing of the message until a new message arrives with a lower VRT. Since this is a probabilistic model, the fact that LVRT >> EVRT does not guarantee a message will arrive with a VRT < LVRT, but the possibility is likely.

Every time a task processes a message with a new virtual receive time, NVRT, EVRT must be recalculated and the task is sorted back onto the task queue. Our estimate only uses first order statistics; for example, we might use a running average or a weighted running average on message arrival times. Let our current EVRT be $evrt_1$. If a new message is processed with an NVRT > $evrt_1$, it tends to cause the estimated average message interarrival time, AT, to increase. The recalculated $evrt_2$ (= NVRT + AT) will be higher than if NVRT=$evrt_1$. This allows the task to be resorted earlier on the task queue, since its LEVRT will be less, thus allowing an earlier execution of its messages. Thus NVRT drives AT (average time between messages), which in turn drives EVRT (= LVT + AT), which then drives the task queue sorting variable LEVRT (= LVRT − EVRT).

4.9.2 Estimated LVRT or EVRT

This method is an attempt to correct for wide deviations in message arrival times. The EVRT strategy chooses the partition which has the lowest LEVRT; an idle processor relocating to this partition will be least likely to cause a backup. In each group's task queue, the task at the front of the queue has the smallest LEVRT, which is also the LEVRT of the partition. EVRT scheduling gives more processing power to the partitions that are least likely to back up.

First a processor from an idle partition that is seeking work retrieves the task queue of each group on the group list. Then the processor relocates itself to the partition with the lowest LEVRT. If none of the partitions have any messages to be processed, then the processor suspends itself in its original partition.

A potential problem is that the EVRT strategy takes additional time in calculating the EVRT of the next message by using a running average. Another is that the EVRT scheme may not perform well in short simulations that have not reached steady state. One fundamental difference between LVRT and EVRT scheduling is that the task queue is now sorted in terms of LEVRT, which is designed for smooth, steady state simulation behavior.

4.9.3 Changes to Forward Message List

In our implementation of the EVRT scheme, a very simple estimate is used. The statistic that is kept is the average time (AT) between successive processed messages calculated as a running average. EVRT is simply the average time (AT) added onto the virtual time (LVT) of the task. The *unprocessed message queue* remains the same as before, but the *processed message queue* is changed. Normally the *processed message queue* contains a list of the processor's messages which have been processed, but in this method, each message is replaced by a pair of the message and the current running average (AT) of the messages that were processed up to that point.

Every time a message is processed, a new running average is calculated, with the message and the average being added onto the *processed message queue.* In this manner backups will work properly since AT is correctly saved in the message list and will be properly restored after backup. AT is used to re-calculate EVRT every time *add-task* or *update-task-queue* are called.

4.10. Continuous Dynamic Repartitioning

A potential problem with all of the dynamic repartitioning strategies is that they do not continuously relocate processors. If a processor is sent to a partition, the processor does not leave the partition until there is no work left. This allows some partitions to get far ahead of others, and the other partitions that are behind could likely cause the partitions that are ahead to back up. Worse yet, a processor that suspends in a given partition because it could find no work anywhere will not be activated until work appears again in that partition, making it unavailable before that time to pick up work elsewhere in the system.

A possible solution to the first problem is to use continuous dynamic repartitioning. Here processors look for work after processing only one task. Now each processor will look for a new partition regardless of the work load in the original partition. As before, if there is no work in any partition, the processor suspends itself in its original partition. This strategy was examined using the lowest LVRT algorithm to determine the new partition; the processor relocates to the partition that is the farthest behind in LVRT.

4.11. Summary

This chapter introduced all of the different scheduling schemes: non-partitioning, partitioning, and dynamic repartitioning. Non-partitioning was used only for base line experiments to determine the advantages of partitioning, while partitioning was likewise used only as a base line to more easily see the advantages of load balancing in dynamic repartitioning. The dynamic repartitioning strategies that were introduced included fixed group list, circular group list, longest task queue, lowest LVRT, EVRT and continuous dynamic repartitioning. The first five methods differed only in the way a relocating processor chose a new partition. The EVRT scheme used statistical means to schedule processor task assignment. Continuous dynamic repartitioning involved possible processor movement after each execution of a task, and used the lowest LVRT to decide which partition to relocate to. In Chapter 6's extreme experiments, each scheduling policy will show off its best simulation operating domain.

V. SIMULATION DEPENDENT SYSTEM

5.1. Introduction

The simulation dependent portion specifies the details of the specific simulation that is to be performed. For each simulation a set of procedure classes must be defined for each different type of task within that simulation. These tasks are then instances of the different procedure classes. Several test simulations were implemented to demonstrate the robustness of the simulation independent system and to test the different scheduling policies. These simulations provide many procedure classes that may be interchanged and reused in other simulations. In this way these simulations provide utilities for building additional larger simulation applications.

5.2. Queueing Simulation

A simple queueing simulation consisting of five different classes of tasks is implemented. These tasks are:

1. Sources
2. Servers
3. Queues
4. Customer Statistics
5. Queue Statistics

The sources create the customers, the servers serve them, queues hold the customers waiting for service, customer statistics gathers data, and queue statistics monitors the number of customers in the queue. Each of these tasks has a message processing procedure which processes any possible input message. The connection of processes in a queuing simulation is illustrated in Figure **5.1**.

The source task creates customers at user specified interarrival times (IAT) or at

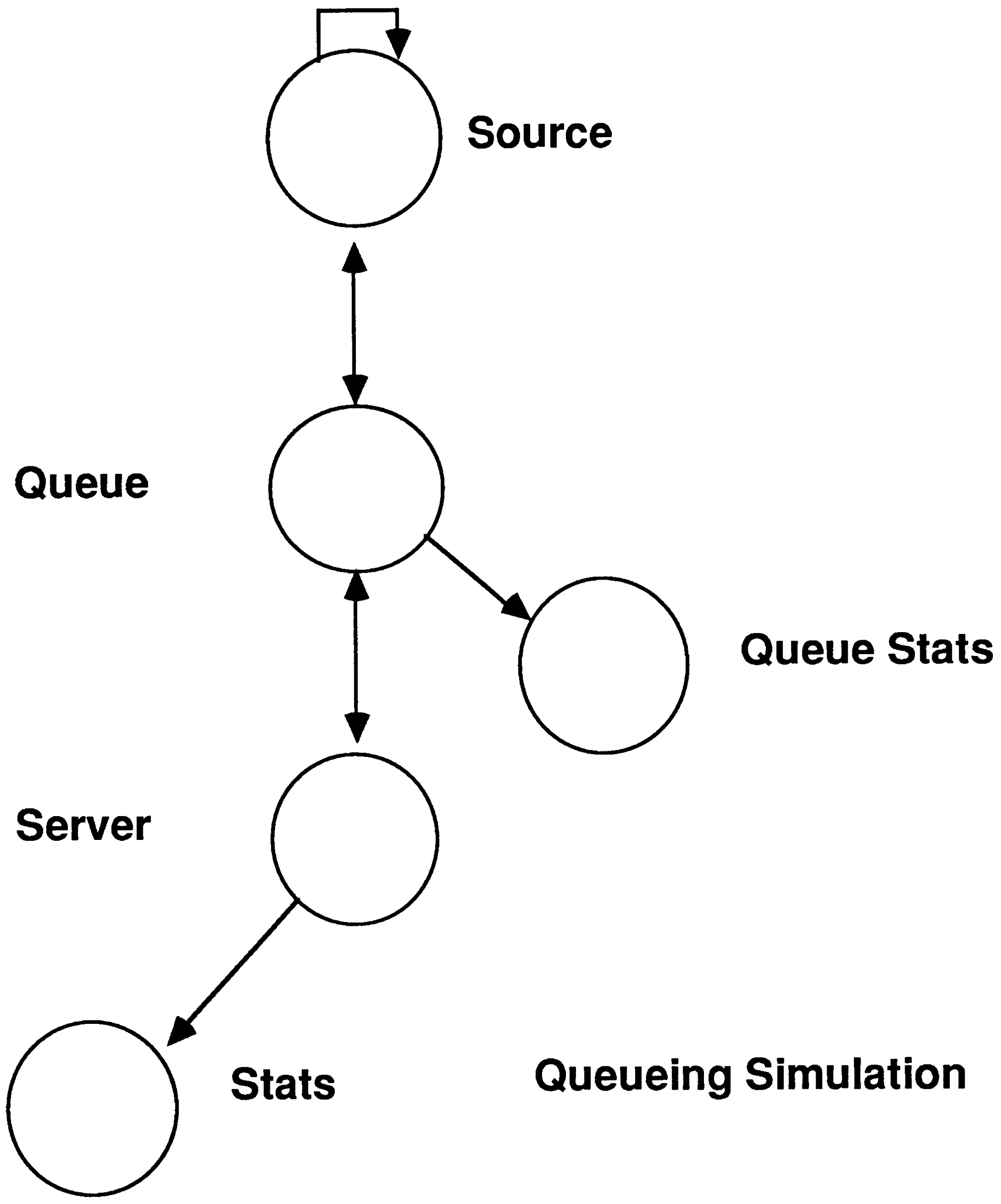


Figure 5.1 Queueing Simulation

random IATs determined by a user specified probability density function. The source creates a customer by sending a message to the queue task with this message's VRT equal to the source's LVT. Next, the source sends a message back to itself requesting that a new customer be created at VRT= LVT+IAT. The source continues to create customers at specified IATs until either the maximum number of creations or the maximum time has been exceeded.

The server task processes customers at user specified service times or at random service times determined by a user specified probability density function. The server task receives customers in the form of messages from the queue task. When a server receives a customer it adds the service time to the VRT of the message to determine when the customer is finished being served. This is now the VRT of two messages that the server task sends: one to the queue task and one to the customer statistics task. The message to the queue task tells the queue when the customer that it had sent has finished being served. The message to the customer statistics task includes the total amount of time that the customer was in the system. The total time consists of the wait time in the queue and the service time.

The queue task receives messages from the source task or the server task. Messages from the source task indicate that a new customer has arrived into the queue waiting to be serviced. Messages from the server task indicate that the server has completed serving a customer and is now free. The queue task keeps a list of free servers in its environment.

When a customer arrives, the queue task first checks to see if there are any free servers. If there are free servers then the customer is immediately processed by sending a message to the server that a customer is ready. The VRT of this queue message is the same as the VRT of the source message, since no time should have elapsed if a server is free.

Alternatively, if a server is not free, then the arrival time (AT) of the customer is placed on the queue. Later, when a server becomes free, this initial arrival time is sent to the server so that the total time the customer spent waiting and being served can be determined.

When a message arrives from the server, the queue task checks to see if there are any customers on the queue. If there are, then the queue task removes a customer from the queue along with its associated arrival time (AT). The wait time of the customer is then VRT − AT, where this VRT is from the server's message. Finally, if the queue is empty then the server is added onto the end of the free-server list.

The customer statistics task receives messages from the server stating the time that a customer waited to be serviced and the time that a customer required to be serviced. This module collects statistics on the wait-service time, this variable being defined to be the sum of the waiting time and the service time. When the simulation is complete, the customer statistics task prints, among other things, and the mean and variance of the wait-service time.

The queue statistics task receives messages from the queue task stating the length of the queue at a particular instant. When the simulation is complete, the queue statistics task calculates the mean length of the queue and the maximum length of the queue, among other things, and then prints these results.

5.3. Circuit Simulation

Here we implement a circuit simulation in which wires and circuit elements are represented by tasks. The simulation consists of the following six tasks:

1. Wires
2. Inverters
3. And-Gates
4. Or-Gates
5. Drivers
6. Probes

Wires are used to connect the circuit elements. Each circuit element has either one or two inputs and one output. The driver and probe tasks are special in that the driver has only outputs and the probe has only inputs. This is illustrated in Figure **5.2**.

Each wire can be in one of two states and is considered to have an input end and an output end. The wire's signal value is determined by the element connected to the its input end. The wire also keeps a list of all circuits connected to its output.

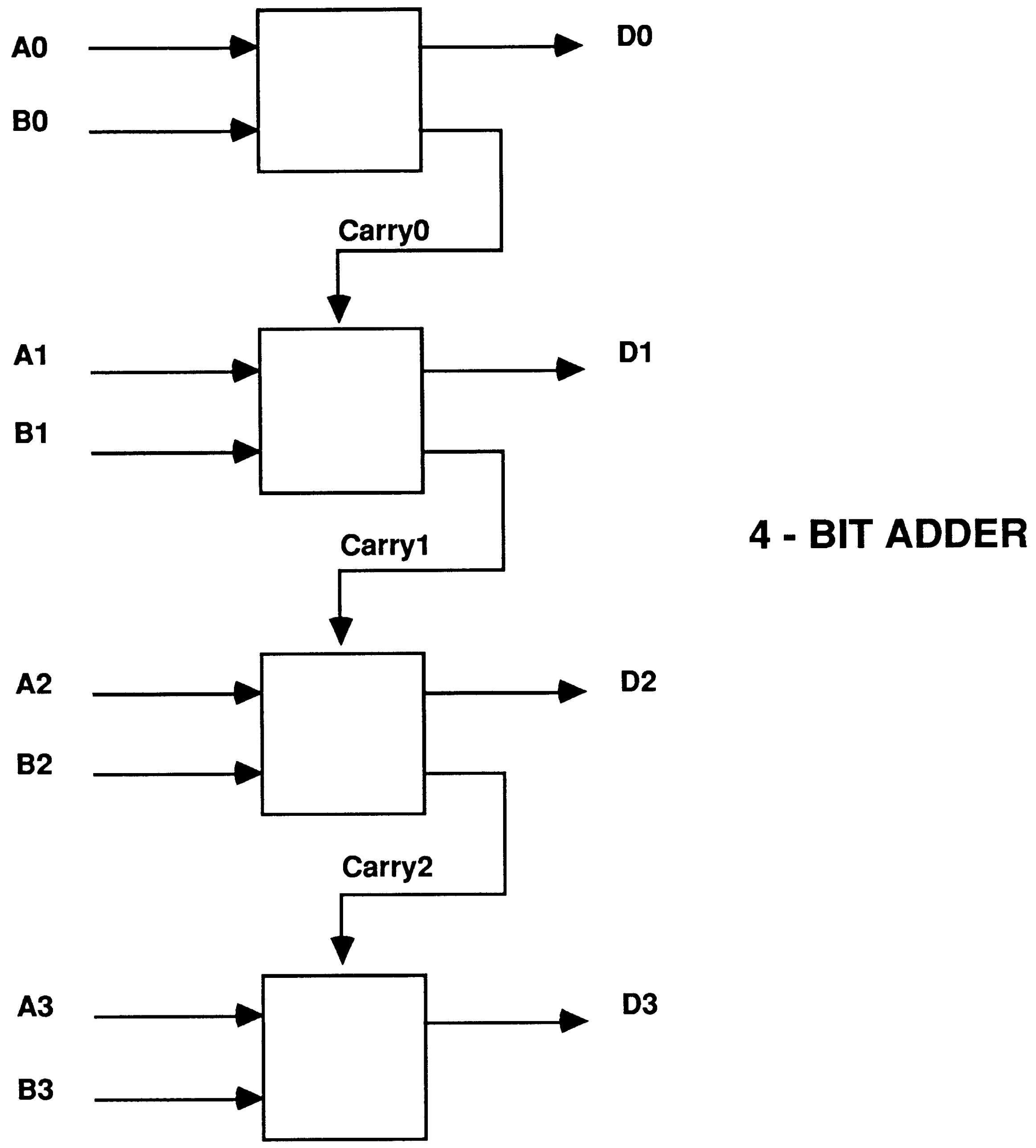


Figure 5.2 Circuit Simulation

When the wire task is first initialized, each circuit element connected to its output is sent a message at time zero containing the initial signal value of that wire. In this way initialization is propagated throughout the circuit. After initialization, changes only occur when a new message arrives at the wire.

When a new message arrives at the wire, the wire first checks to see if its signal value has changed. If it has, then the wire sends this change on to all the circuit elements connected to its output. This message is sent without a delay; in other words, the VRT of the outgoing message is the same as the VRT of the incoming.

The inverter task takes the signal value of its input wire and then sends its complemented value over its output wire after an inverter delay (ID). The signal value of the output wire (outval) is a variable saved in the inverter's environment.

Whenever the inverter receives a message (with a VRT=vrt_1), which requires a change in the signal value of its output wire, then a new message is sent to this output wire with a VRT=vrt_1+ID. The value of the output wire is updated in the outval variable.

The and-gate task takes the signal value of its input wires (a1 and a2) and then sends their logical AND to its output wire after applying an and-gate delay (AD). The signal values of the output wire (outval) and the two input wires (a1val and a2val) are variables saved in the and-gate's environment.

Whenever the and-gate receives a message (with VRT=vrt_2), it stores the new input value in either a1val or a2val depending upon which input wire sends the message. If the input comes from a1, for example, then the new value for the output is the logical AND of a2val and the new a1 signal value. If the gate's outval is thus changed, the change is sent from this gate to its output wire with a VRT=vrt_2+AD. The value of the output wire is updated in the outval variable.

OR gates are just like AND gates except that they perform the logical OR function rather than the logical AND.

The driver task changes the signal value of its output wire at a pre-specified time determined by the user. It uses two lists, one containing the set of inputs to its output wire, and its counterpart containing the virtual times at which those inputs were to take place. Drivers are used to start and drive the simulation.

Probe tasks display changes in the signal values of wires to the user. When a probe receives a message from a wire task, it prints the name of the source task, the VRT of the message, and the new value which was sent. Probes display intermediate results and print out the final values of the simulated circuit's output wires.

5.4. Butterfly Network Simulation

Butterfly networks of 4, 8, and 16 input nodes are implemented. The network topology is exactly like the butterfly network used to implement FFT (Fast Fourier Transforms). With 2^n input nodes, there are n stages, each node having two inputs and two outputs. From any input, one can reach any destination in n transitions. Again, this is a very simple topology, and one can easily build large butterfly networks using only three basic tasks as building blocks:

1. Nodes
2. Drivers
3. Probes

This topology, illustrated in Figures **5.3** and **5.4**, is interesting because it allows the study of several different partitioning strategies.

The node tasks control the routing of the customers through the system. Upon receiving a new customer, the node task sends that customer along a predetermined route to the destination probe. All customers enter through one of the 2^n input nodes and propagate through n transitions towards the destination probe. Each destination probe represents a number from 1 to 2^n.

Let ND be the nodal-delay for any customer traversing a node, and let CD be the conflict-delay for consecutive customers with the same original VRT. Let r_i be the VRT of the ith incoming customer, and t_i be the virtual time at which the ith customer leaves the node. Then t_i should be given by $t_i = \max(t_{i-1}, r_i) + CD$. The VRT for the customer's arrival at the next node should be $t_i + ND$.

Routing is determined by the data field of the message representing the customer. The message's data field is a list of binary digits representing the predetermined routing. If the first element of that field is 0, then the message is sent to the first output node

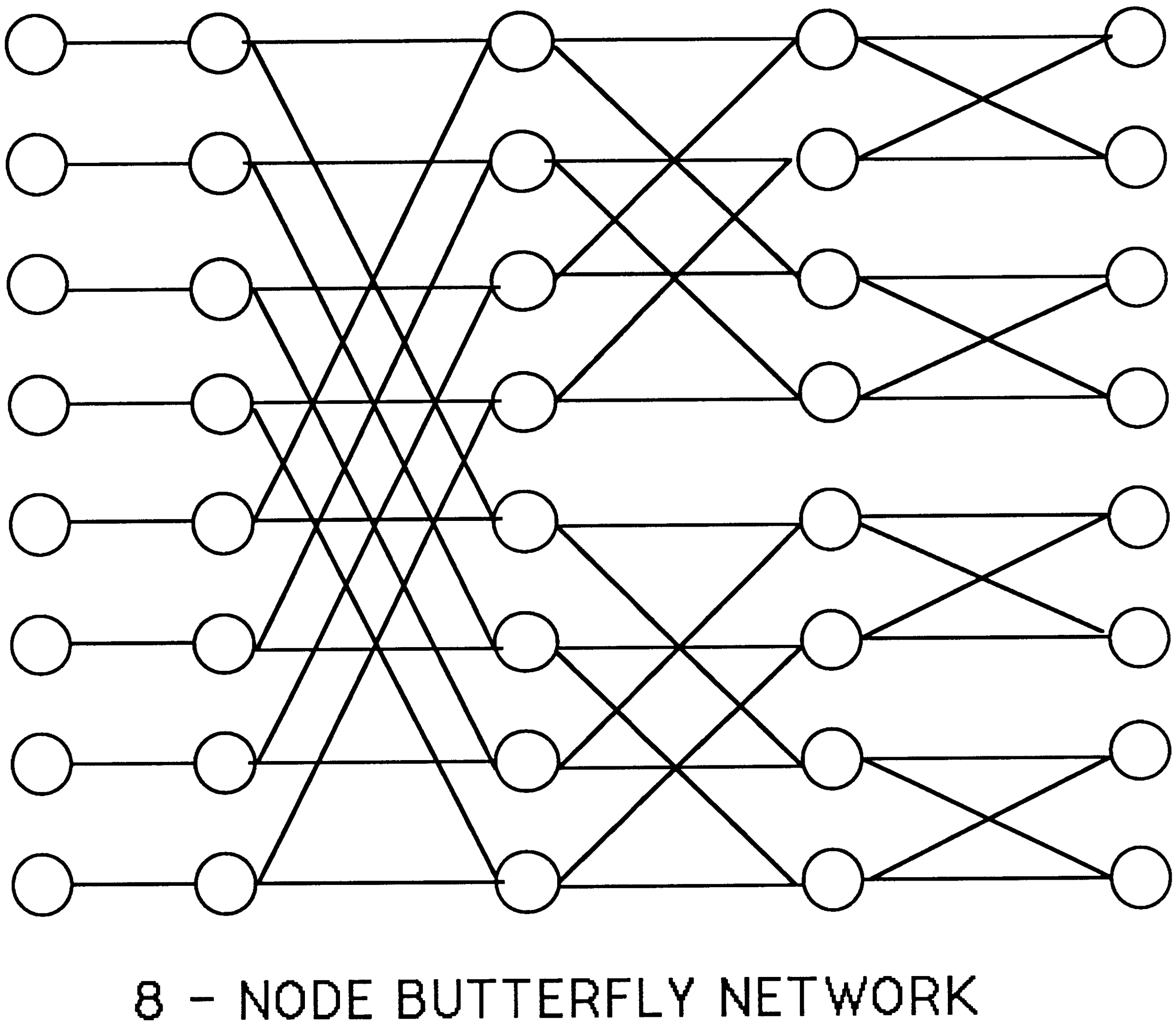


Figure 5.3 8-Node Butterfly Network Simulation

(or probe) with a new data field without the first element. If the first element is 1, then a message is sent to the second node having the revised data field. Since we use a butterfly network configuration, this routing method guarantees the customer's arrival at the correct destination probe by traversing only n network nodes. When the message's data field becomes empty it indicates that the customer has reached the destination probe, and subsequently that probe prints both its own name and the VRT

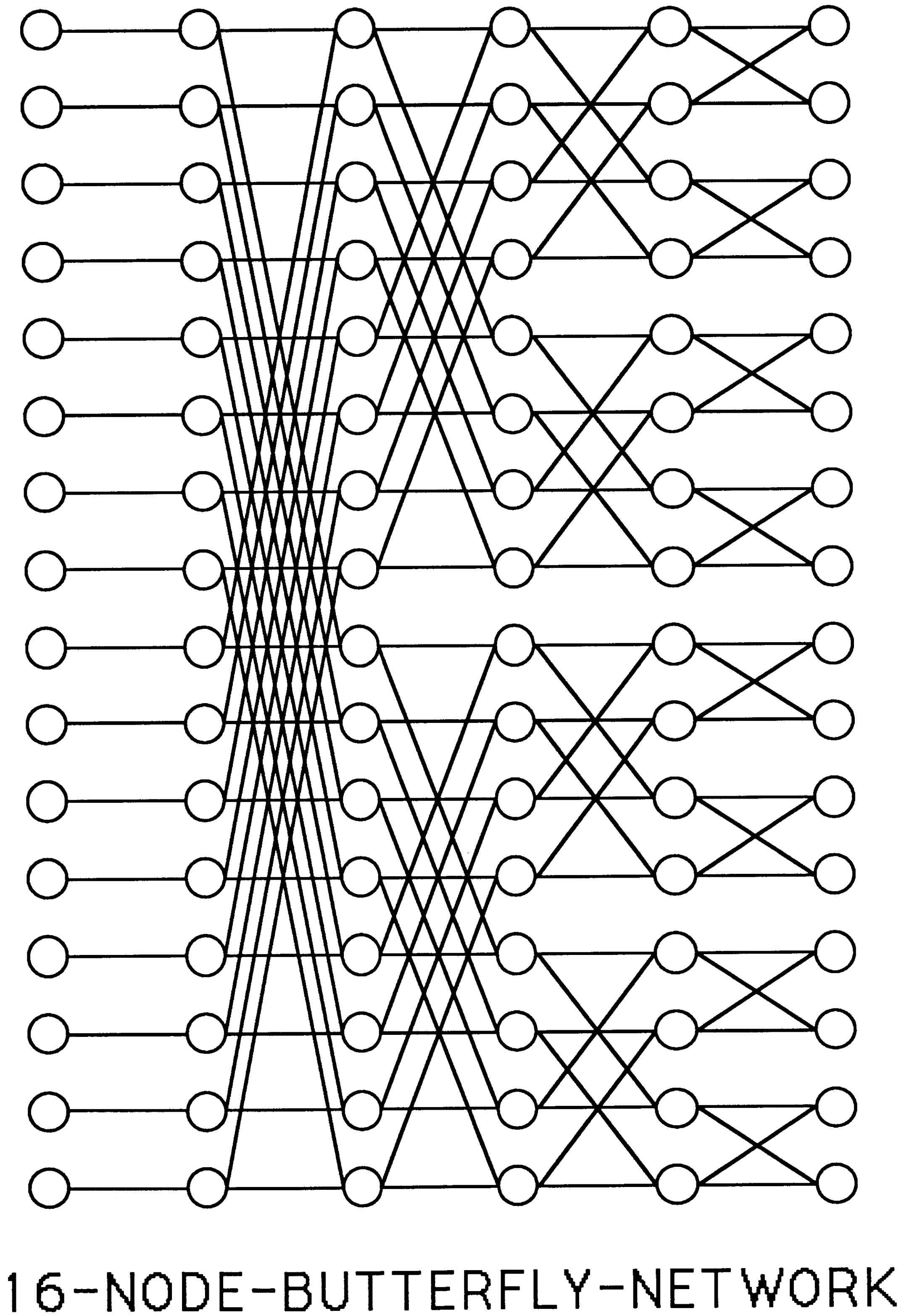


Figure 5.4 16-Node Butterfly Network Simulation

of the message. In this way the customer is said to have exited the system.

Driver tasks send customers across the network toward specific locations at prespecified user determined times. Each driver has a start node, where all the customers it creates are initially sent. A customer departs this start node and then travels along a predetermined route specified by the data within its message. As before, the driver task uses two lists, one part containing data specifying the route and the destination probe, and the second part containing the VRT at which the message is to be launched from the driver.

5.5. Partitioning

In this section we describe the partitioning methods for a 16-input-node butterfly network. Since the user determines how tasks should be partitioned, it is important to know what type of partitioning method performs best. We will show later that the method of partitioning can make a big difference in the performance of our simulation. We ran experiments involving different partitioning methods on 16-input-node butterfly networks. The different partitioning methods used in the experiments were:

1. Horizontal Partitioning (with eight (H_8) and 16 (H_{16}) partitions)
2. Random Partitioning (with eight (R_8) and 16 (R_{16}) partitions)
3. Minimum Communication Partitioning (with eight (MC_8), twelve (MC_{12}), and 24 (MC_{24}) partitions)
4. Vertical Partitioning (with six (V_6) partitions)

5.5.1 Horizontal Partitioning

We partition simulations performed on a 16 input-node butterfly network into either eight or 16 equal groups (partitioning methods H_8 and H_{16} respectively). These partitioning methods are shown in Figures **5.5** and **5.6**.

In the 16-partition case, each partition comprised a horizontal line of tasks, so that a message could be sent from one task to the next within the partition and then exit the system after four stages. Each partition consisted of one driver task *d*, four node

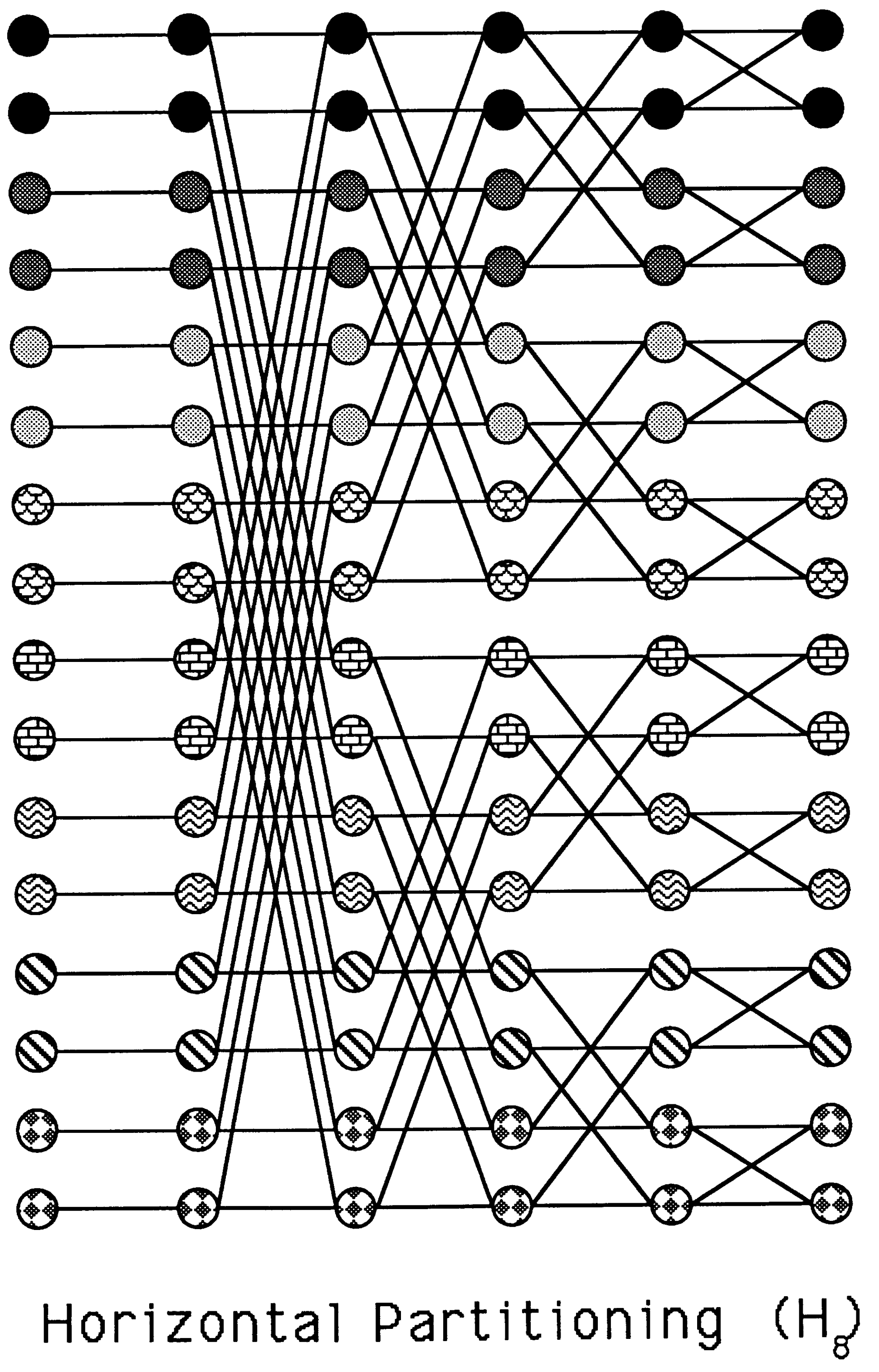


Figure 5.5 Horizontal Partitioning with Eight Partitions H_8

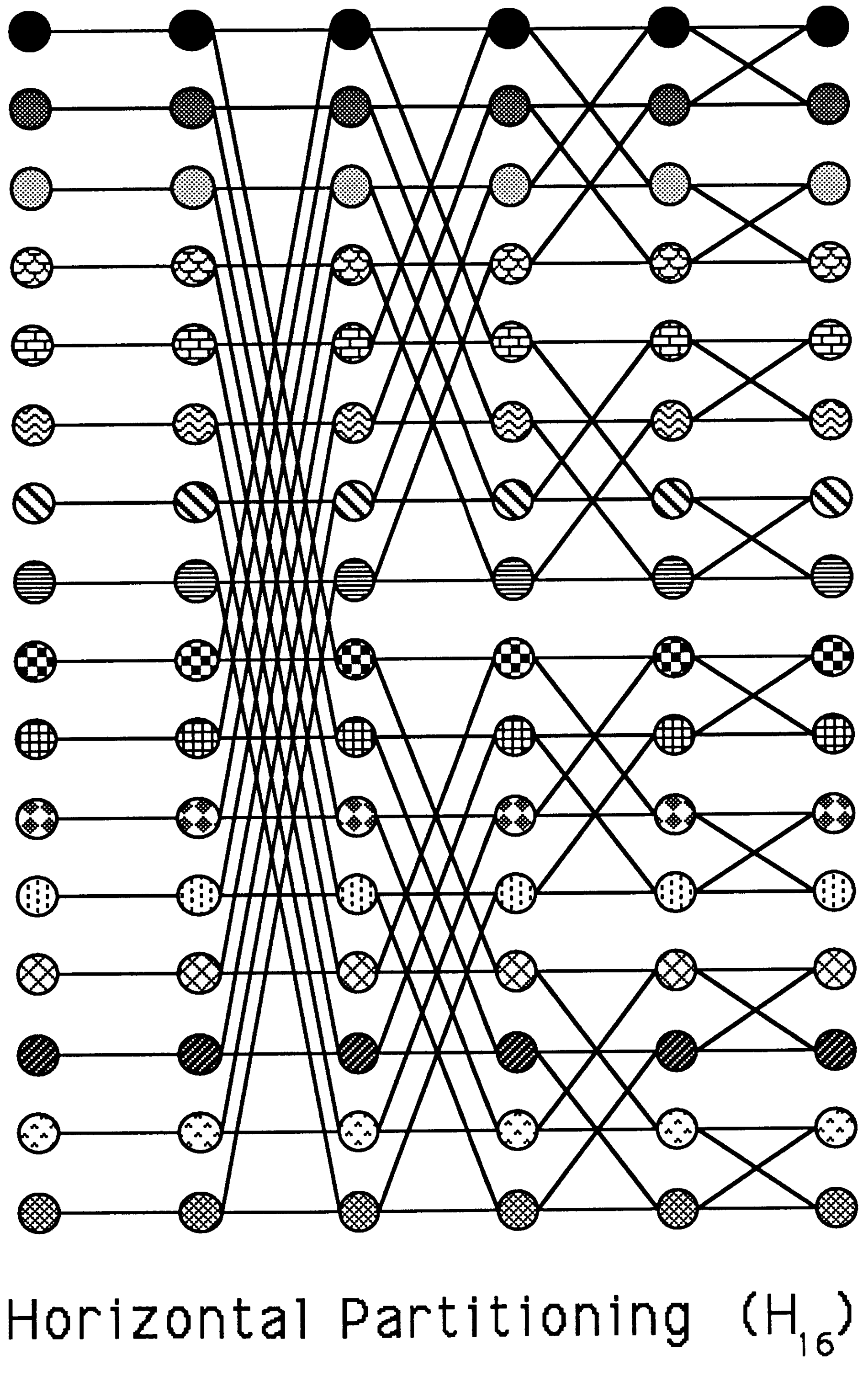


Figure 5.6 Horizontal Partitioning with 16 Partitions H_{16}

tasks (n_1, n_2, n_3 and n_4) and one probe task p. The flow of messages through the partition started at d and then went to n_1, n_2, n_3, n_4, and finally on to p. The probe task p acted like a node task with no subsequent node task.

In the eight-partition case, each partition consisted of two horizontal lines of tasks. In other words, each H_8 partition contained of two neighboring H_{16} partitions.

5.5.2 Random Partitioning

Six different random partitions were used on a 16-input-node butterfly network. They were:

1. R_{8a} with 8 partitions
2. R_{8b} with 8 partitions
3. R_{8c} with 8 partitions
4. R_{16a} with 16 partitions
5. R_{16b} with 16 partitions.
6. R_{16c} with 16 partitions.

These partitioning methods will be shown in the Appendix. Partitions belonging to R_{8a}, R_{8b} and R_{8c} were made to contain 12 tasks each, whereas partitions belonging to R_{16a}, R_{16b} and R_{16c} had 6 tasks each chosen randomly using a uniform random variable. These tasks were either nodes, drivers or probes.

5.5.3 Vertical Partitioning

Vertical partitioning on a 16-input-node butterfly network consisted of 6 groups partitioned vertically (V_6) with respect to the flow of messages. Each partition consisted of 16 tasks. The first group consisted solely of drivers. The second, third, fourth, and fifth partitions consisted solely of nodes in a vertical grouping. Finally the last partition consisted solely of probes. This partitioning method is shown in Figure 5.7.

5.5.4 Minimum Communications Partitioning

Minimum communications partitioning grouped our 16-input-node butterfly network into groups of two-input-node and-four input-node butterfly networks. In this

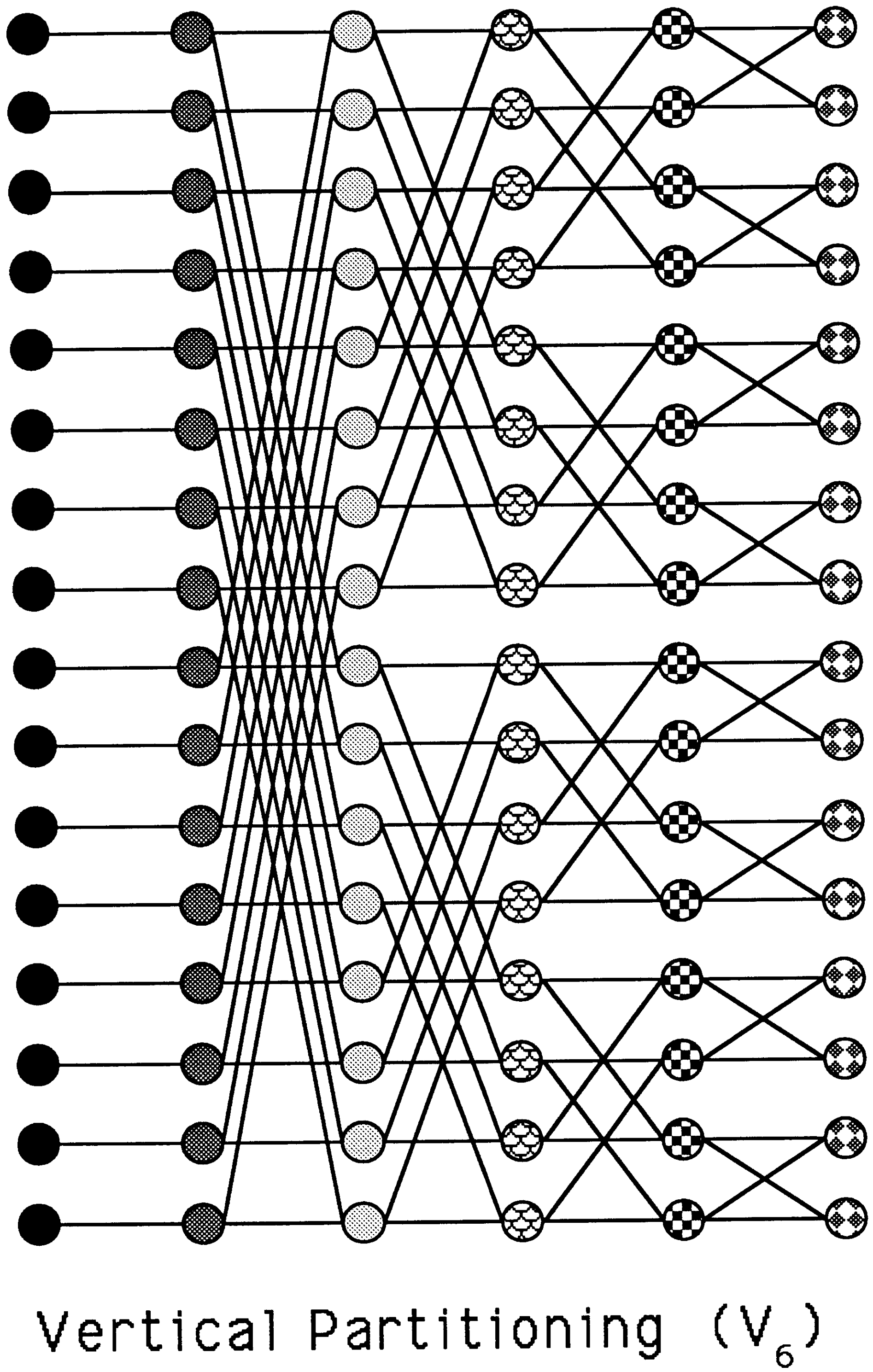


Figure 5.7 Vertical Partitioning with 16 Partitions V_6 (laura)

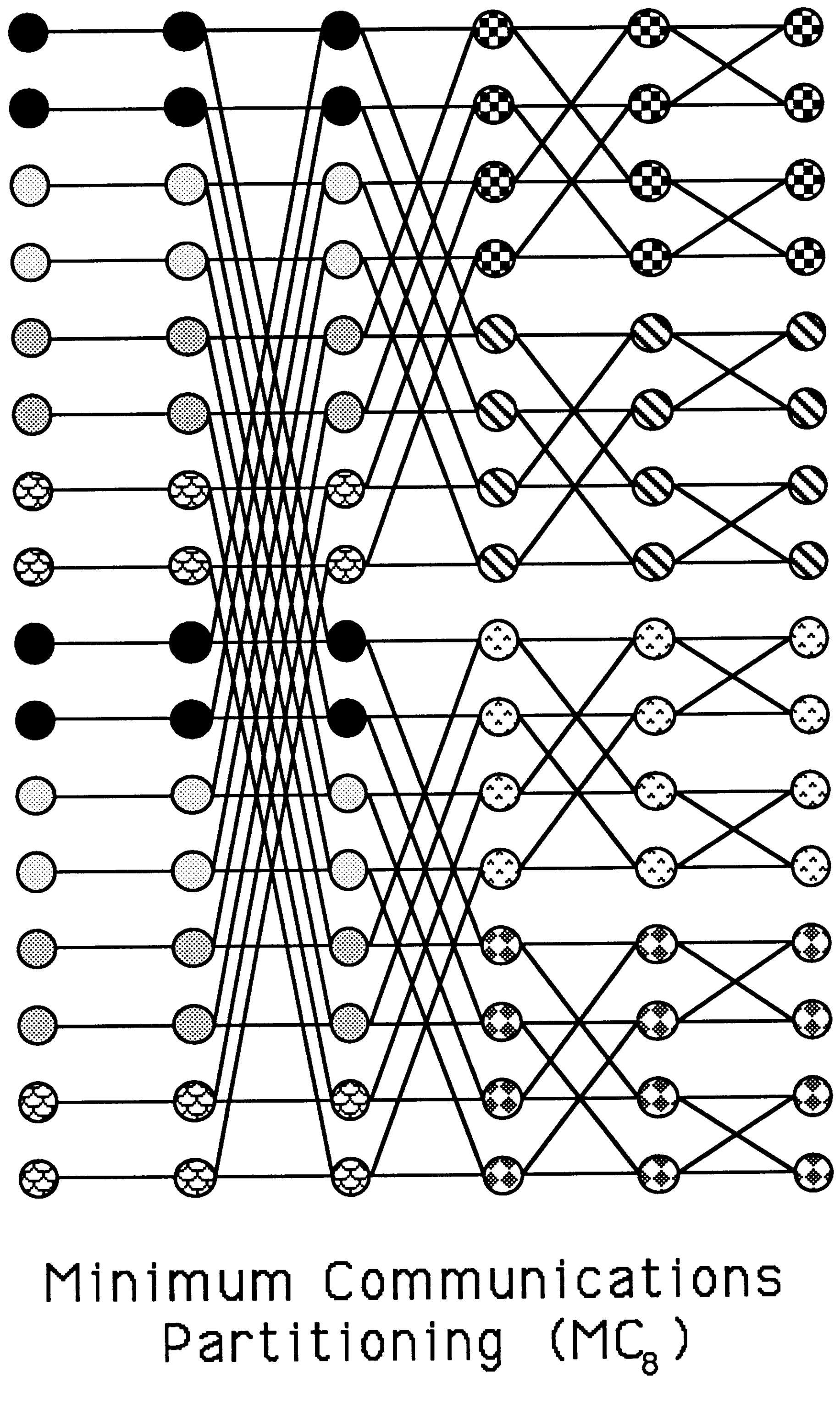


Figure 5.8 Minimum Communication Partitioning with 8 Partitions MC_8

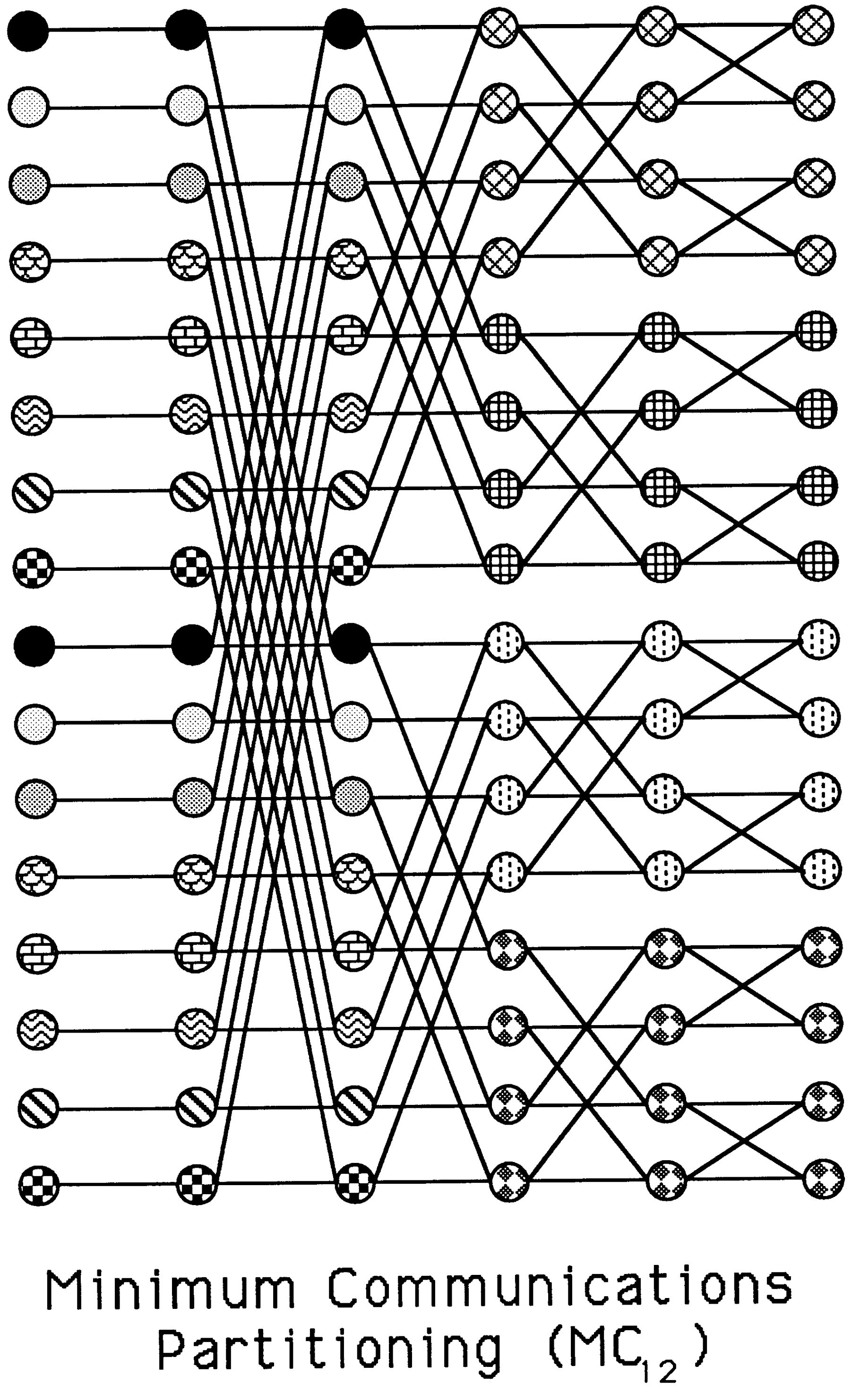


Figure 5.9 Minimum Communication Partitioning with 12 Partitions MC_{12}

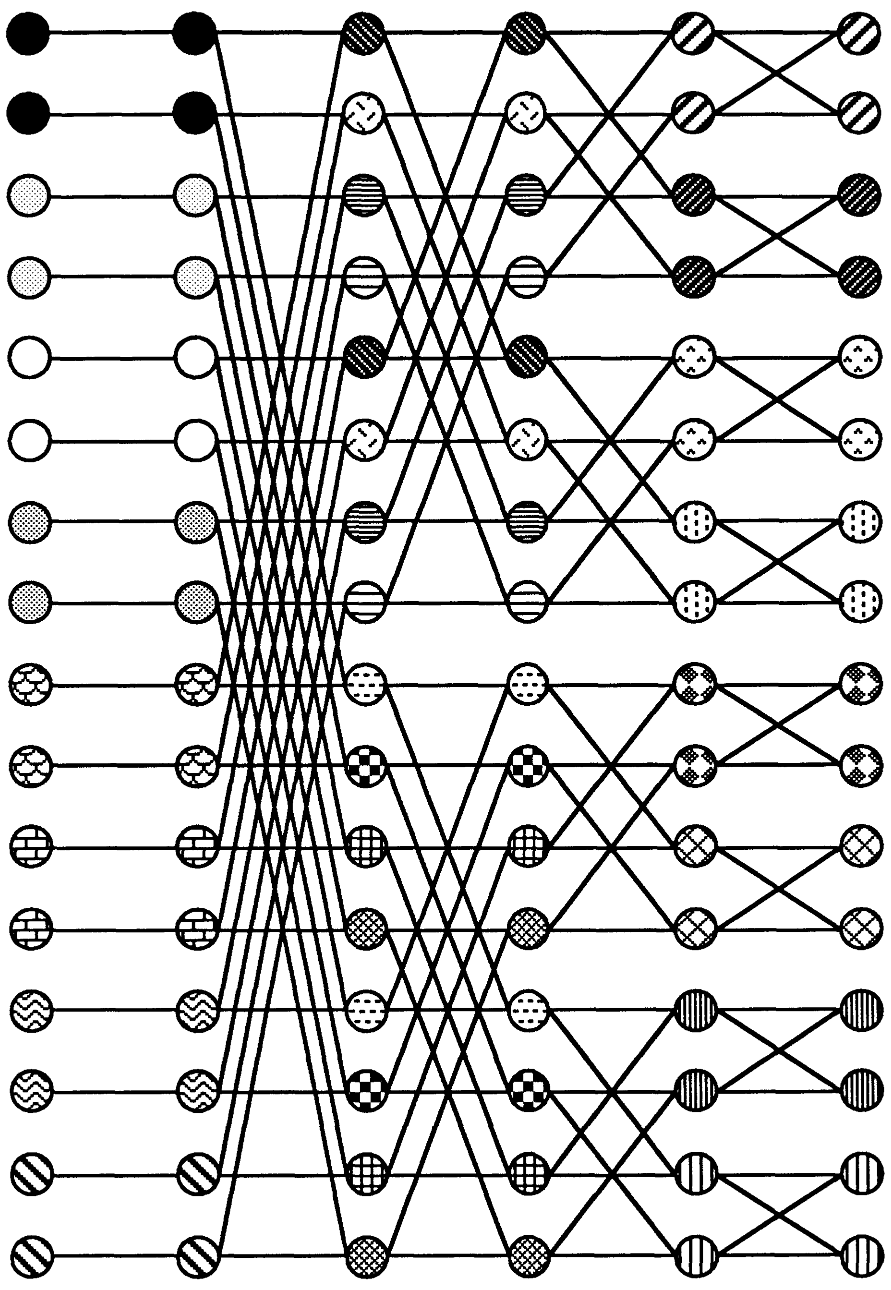


Figure 5.10 Minimum Communication Partitioning with 24 Partitions MC_{24}

manner minimum communications partitioning involved forming groups that minimized the number of inter-group links.

Three different minimum communication partitioning methods were tried on a 16-input-node butterfly network. They were:

1. MC_8 with 8 partitions,
2. MC_{12} with 12 partitions,
3. MC_{24} with 24 partitions.

Each was partitioned on a 16-input node butterfly network, as illustrated in Figures **5.8**, **5.9**, and **5.10**.

In MC_8, four of the partitions consist of two two-input-node butterfly networks with drivers connected to the 4 input nodes. These four partitions each consisted of 12 tasks combining two of the two-input-node butterfly networks with drivers. The last four partitions consisted of four-input-node butterfly networks, again resulting in a total of 12 tasks.

In MC_{12}, eight of the partitions consisted of two-input-node butterfly networks with drivers connected to their two input nodes. These partitions consisted of 6 tasks each. The other four partitions consisted of four-input-node butterfly networks containing 12 tasks each.

Each of the MC_{24} partitions consisted of four tasks. Eight of the partitions were simple driver-to-input-node networks, whereas the other 16 were two-input-node butterfly networks.

5.6. Summary

The variety of simulations discussed in this chapter demonstrates the wide applicability of our system. The idea of defining procedure classes for each task in the simulation proved to be essential. Within these simulations two procedure classes, Probes and Drivers, were reused. Probes, for example, were used in three of the simulations: circuits, butterfly network, and grid network. It was this idea of the user being able to write tasks as modular, interchangeable procedure classes which gave our simulation dependent structure its flexibility.

Another simulation that could be modeled by using some of the procedure classes just discussed would be a computer network. Its network nodes could be modeled by using a modified network node procedure class, and the queues which hold the packets entering these nodes could be modeled by a modified queue procedure class from the queueing simulation. Even though our system is robust, one cannot say that we have covered all types of discrete time event-based simulation. For example, traffic flow cannot be modeled with the procedure class building blocks in this chapter (but can be modeled within the structure of our system if suitable building blocks are defined).

VI. EXTREME EXPERIMENTS

6.1. Introduction

This chapter presents a set of extreme case experiments, each of which highlights a specific scheduling strategy. The chapter also supports some of the claims made in Chapter 4 about potential problems with each of the different scheduling strategies. We proceed through a series of experiments, named Example 1, Example 2, etc., that highlight performance differences among five distinct dynamic repartitioning strategies. These experiments provide measurements of the strengths and weaknesses of each strategy and also show by example that no strategy is always the best. We focus on how each strategy attempts to avoid the processing of backup-inducing messages with high VRT.

6.2. Performance Measurements

We analyze performance in two ways:

1. Processing time
2. Number of backups.

Measuring the number of backups is important because in our simulations, the task's environment is relatively small, consisting of one variable; thus individual backups are relatively cheap. If one were to increase the number of variables in the environment, then backups would become expensive and could influence processing time. We also measure performance in terms of backups because there is a large and ever-present variation in Concert's processing time due to random occurrences of garbage collection.

Processing time consists of several components. Figure **6.1** is a time line representing one simulation. The times t_0, t_1, t_2, and t_3 are the times of particular events. t_0 is the start time of the simulation; t_1 is the time when all initializations are complete;

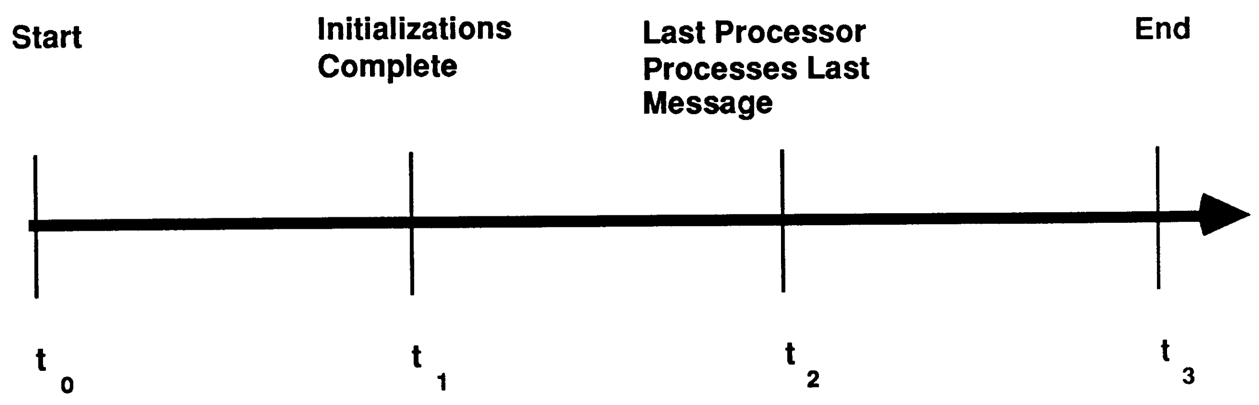

Start
Initializations Complete
Last Processor Processes Last Message
End
t_0
t_1
t_2
t_3
Real-Time Line of a Simulation

Figure 6.1 Real-Time Line of a Simulation

t_2 is the time when the last processor processes the last message; and t_3 is the time when the simulation is complete.

The types of processing times can be split into different categories:

1 . **Startup Time:** The time for all initializations, (as described in Section 3.8) to occur. This is the duration between t_0 and t_1 on the time line.

2 . **End Time :** The time, after the simulation is complete, needed to calculate statistics and print results. This is the duration between t_2 and t_3 on the time line.

3 . **Wait Time : Total wait time** is the aggregate time that processors spend waiting in one simulation run. This time is entirely between t_1 and t_2. This is computed by recording the times when a processor requests a lock and when a processor receives a lock. The difference between these two quantities represents the time for a single wait of a processor. The total wait time is the sum of these single wait times. **Wait time per processor** is the total wait time divided by the number of processors. Wait time can be broken into three sub-components:

a. **Wait-add-task**: The total waiting time for processors trying to add a task to the task queue.

b. **Wait-next-task**: The total waiting time for processors trying to execute the function (*next-task*), which is described in section **3.3.4**.

c. **Wait-update**: The total waiting time for processors trying to re-sort a task onto the task queue after the task's LVRT changes.

4 . **Suspend Time**: The time per processor that elapses when there is no work for the processor. This is computed by marking the times when a processor becomes deactivated and when a processor becomes activated. The difference between these quantities is the time of a single suspend. The suspend time is computed by summing the single suspend times and dividing by the number of processors.

5 . **Overhead Time**: Wait Time per Processor + Startup Time + End Time

6 . **Processing Time**: The time, less wait time, spent in simulation; this quantity equals Total Run Time − Startup Time − End Time − Wait Time per Processor.

We include suspend time as part of the processing time. This approach can be both descriptive and misleading.

Including the suspend time with processing time could be misleading because, if we include suspend time, a simulation that has processing time PT using n processors would not necessarily have processing time nPT with one processor. This is because not all of the n processors are busy for PT seconds; some of them are suspended due to lack of work. Thus processing time includes some time not spent "processing." However, if we do not include suspend time, a simulation with processing time PT with n processors would have processing time nPT with one processor (if no backups occur).

However, including suspend time in processing time makes sense because in any scheduling algorithm, some of the processors will have no work to do some of the time. We wish to measure this lack of parallelism.

For example, suppose a simulation takes PT1 seconds on a single processor. Suppose we run this simulation on our system with n processors and that no parallelism can be exploited. This would be the case if the simulation were a set of sequential operations that depended on each other. This means that one processor does all of the work while the others are idle. If we subtract suspend time from processing time, then the total processing time would be

$$PT1 - \frac{n-1}{n} \cdot PT1 = \frac{1}{n} \cdot PT1 \ seconds.$$

This would appear that the simulation with n processors ran n times as fast as the simulation with one processor! This makes no sense; therefore, we include suspend time in processing time.

We do not include wait time in processing time because wait time is a measure of task queue contention. Wait time may be reduced by increasing the speed of the system software that implements semaphores and queues which are used in the task-queue locks. This system software is independent of the scheduling algorithm. Therefore, since wait time might be reduced without changing the scheduling algorithm, we do not include it in processing time.

6.3. Best and Worst Case for Unpartitioned Scheduling

An unpartitioned scheme has less overhead than does a partitioned strategy (partitions need to be initialized, synchronized, and executed). An unpartitioned simulation will outperform a partitioned one in the case where the simulation is executed trivially on a single physical processor. This advantage disappears as the number of physical processors is increased. The fundamental reason for this is task queue contention; that is, many physical processors simultaneously contend for a task queue. Because this task queue must be locked by a single processor while the processor is accessing a task, the other processors must then wait.

Example 1 dramatizes this task queue bottle-neck which is present in non-partitioning. Our experiment was run on a 16-node butterfly network. The first half of the experiment combined all of the tasks into one large partition simulating the non-partitioning strategy with a single task queue. All 32 processors were allocated

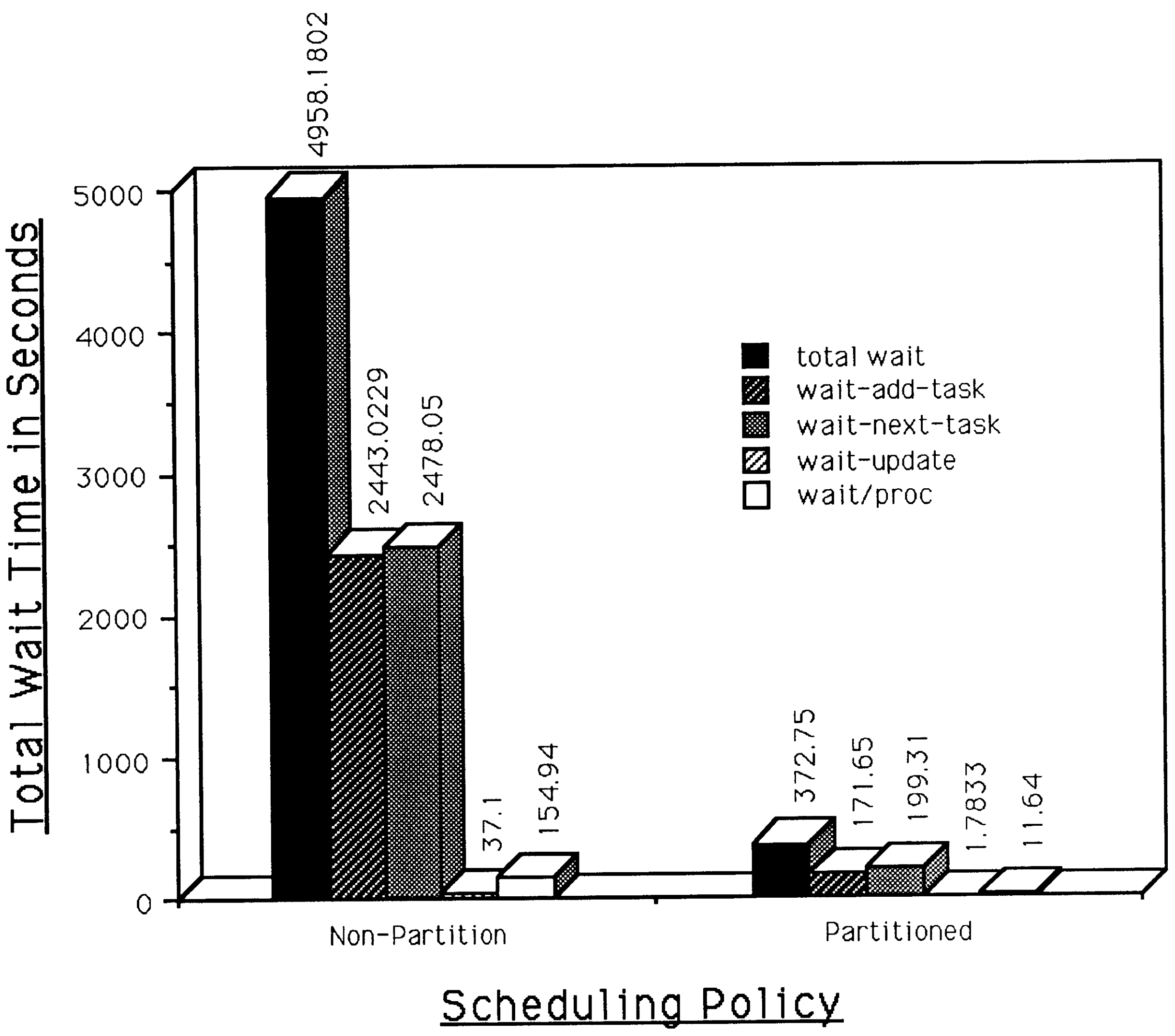


Figure 6.2 Wait time of Non-Partitioning and Partitioning

to the single partition. The second half used partitioning, which partitioned the simulation into groups of tasks, each having its own smaller task queue. Each partition was allocated 4 processors. The partitioning method used was minimum communication with 8 partitions (discussed in Chapter 5); Figure **5.8** illustrates this partitioning method applied to a 16-node butterfly network.

Figure **6.2** shows the different amounts of waiting time, in seconds, for both the non-partitioning and static partitioning strategies. The wait time experienced in the partitioned case is $\frac{1}{14}^{th}$ that of the unpartitioned, as shown in Figure **6.2** . The strong indication is that partitioning reduced the total simulation time by reducing the wait time per processor by a large factor.

Our implementation of message and task queues used a sorted list. This method may not be optimum to specific simulations. For example if one can characterize the frequency of different messages' VRTs, then it may be possible to implement the queues more efficiently. On the other hand, our implementation was designed to be general; therefore, we used a non-simulation-specific implementation of queues.

6.4. Best and Worst Case for Partitioned Scheduling

The static partitioning strategy will outperform dynamic repartitioning in cases where there is very little need for processor movement (if the programmer knows his problem well and has initially assigned his processors well). In static partitioning when a processor finds no work within its partition, it suspends itself, while in dynamic repartitioning there is the additional "overhead" of the processor's having to search for work in the other partitions.

Dynamic repartitioning can outperform static partitioning even if there is a good initial assignment of processors, provided the optimal number of processors assigned per partition is a non-integral value. In this case there will be points in simulation time where processor movement will speed simulation execution.

Example 2 contrasts static partitioning against dynamic repartitioning. Our experiment was run on a 16-node butterfly network. The partitioning method used was random with 8 partitions (discussed in Chapter 5); Figure **6.3** shows an example of a random partitioning method with 8 partitions for a 16-node butterfly network. Of the 8 partitions, one was allocated 25 processors while the others received only one processor. In this manner we could demonstrate the usefulness of processor relocation. The first half of the experiment demonstrated the essential trouble with static partitioning: an out-of-work processor would simply suspend itself and waste its valuable cycles. The second half of the experiment showed dynamic repartitioning of processors in partitions with no work to ones with work, depending upon a relocation algorithm (in the

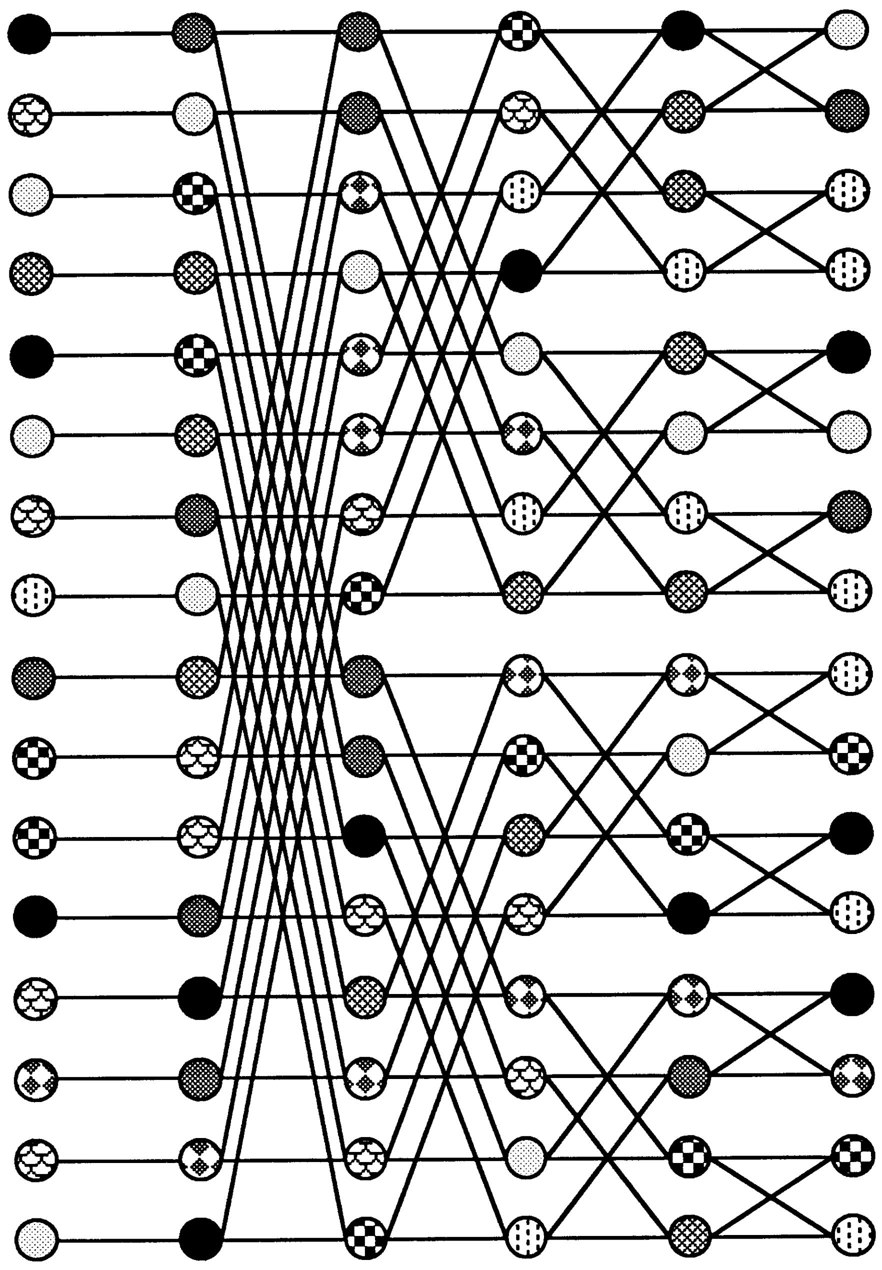


Figure 6.3 Random partitioning with 8 Partitions R_{8a}

next section we will talk about five of these specific algorithms). Thus at the cost of additional overhead, dynamic repartitioning solved (by relocation) static partitioning's central problem of wasted processor cycles.

Figure **6.4** shows the difference in processing time in our experiment between static

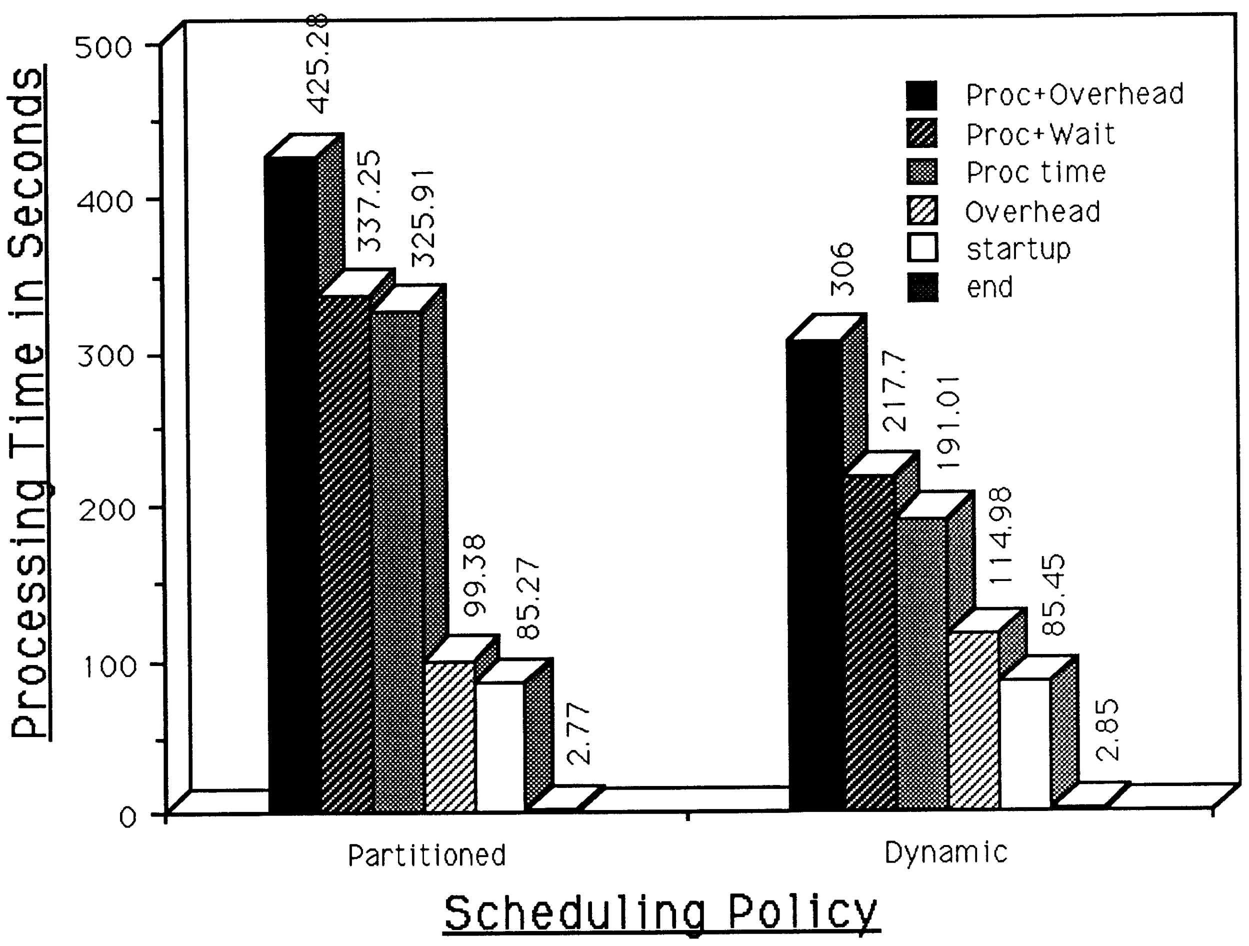


Figure 6.4 Processing time of Static Partitioning and Dynamic Repartitioning

partitioning and dynamic repartitioning. Figure **6.4** shows the different amounts of processing time, in seconds, for both the static partitioning and dynamic repartitioning strategies. Dynamic repartitioning required less processing time than static partitioning in this experiment.

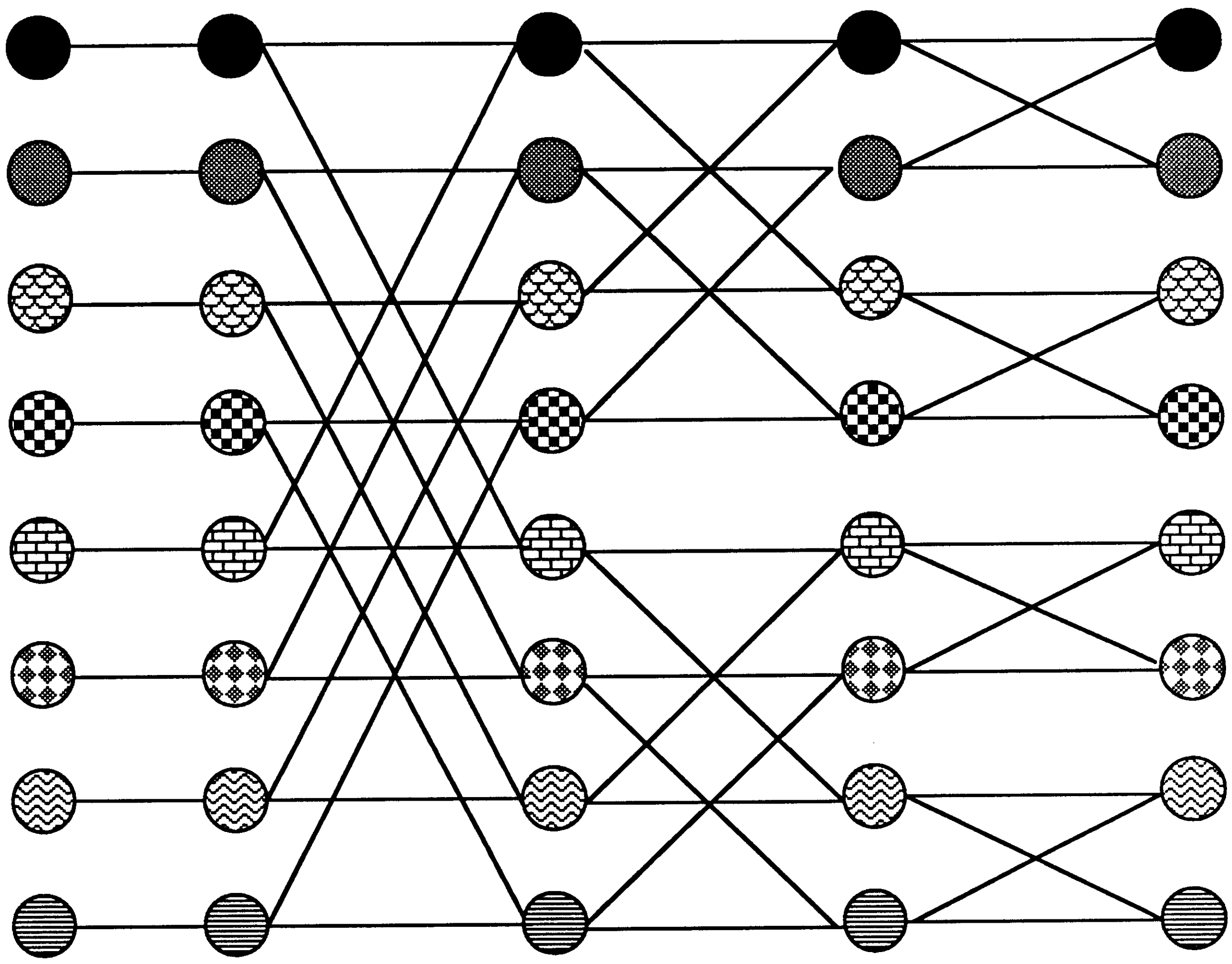


Figure 6.5 Horizontal partitioning with 8 Partitions on a 8-input node butterfly

6.5. Best and Worst Case for Dynamic Repartitioned Scheduling

In the following subsections five different experiments were done, each demonstrating how one dynamic repartitioning algorithm was able to defeat the other strategies. All of the experiments were performed on an 8-input-node, 3-stage butterfly network. Each partition consisted of five horizontally linked tasks: the first a driver, the middle three nodes, and the final task a probe. Although this is not an optimal partitioning strategy for this network, this horizontally partitioned network was the easiest to analyze and besides, optimality of partitioning is not the issue; comparing scheduling policies is the issue. Figure **6.5** shows the topology of the network that was used with its corresponding partitions. The results of other experiments on this 8-input butterfly network using other partitioning methods will be discussed in Section **7.8**.

6.5.1 Group with Lowest LVRT

Figure **6.6** shows the interactions between partitions in Example 3, where the lowest LVRT dynamic strategy outperforms fixed-list, circular-list, and longest task queue dynamic repartitioning. In the figure each labeled node represents a partition. The leftmost column represents sending partitions, while the rightmost enumerates receiving partitions. A link between a pair of nodes represents a message communication path. Each experiment has a sequence of events that triggers a backup. The point of each experiment is to highlight a dynamic strategy which avoids this trigger, and hence performs the best (no backups).

In order to discuss messages from a source to a destination with an associated VRT, we introduce a new notation M_{VRT}^{s-d}. For example, M_{710}^{4-3} refers to a message which originates at partition 4, is destined for partition 3, and has an associated VRT of 710.

In Figure **6.6** backup will be prevented by processing all of the messages originating from partitions 2, 3, and 5 before executing any messages from 1 (processing M_{700}^{1-1} is the backup trigger).

In this experiment partitions 4, 6, 7 and 8 were decoupled, in the sense that they sent messages only to themselves and received no messages from other partitions. If not enough messages were sent through these decoupled partitions, then the processors servicing them would migrate to the other partitions, thus disrupting the experiment. Hence, we keep them busy. Partitions 2, 3, and 5 all send messages to 1 as well as to themselves, as shown in the diagram.

Partitions 2, 4, 6, 7, and 8 begin the experiment with four physical processors each. The group list is ordered (1 4 6 7 8 5 3 2). We say that a partition saturates when there is no work for a free processor at a particular time, and hence that processor will seek to relocate to another partition. Generally, a partition can have 2 or 3 processors

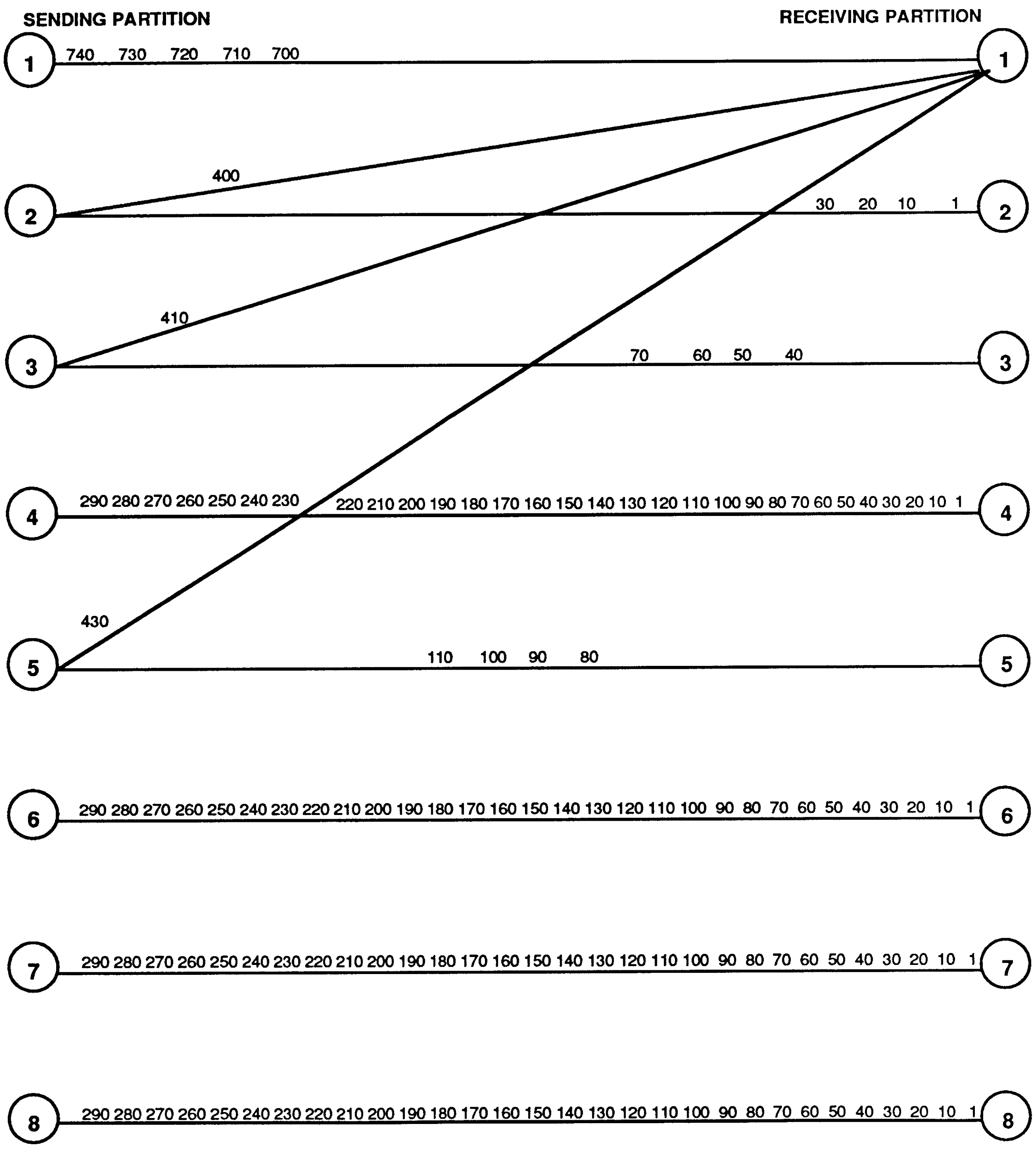


Figure 6.6 LVRT Superior over Other Schemes

running within it before it saturates. This experiment is designed so that partitions 2, 4, 6, 7, and 8 start the experiment saturated and then each of them will soon free a migrating processor. When the simulation begins, each partition has one task on its task queue corresponding to the driver task. The driver task was placed on the task queue during the simulation initialization.

In the lowest LVRT strategy, this migrating processor will first relocate to partition 3. When partition 3 is out of work, then migration will begin to partition 5. The last partition to be activated will be 1 (a partition is activated when a processor begins executing its messages). In this way all of the messages from 2, 3 and 5 destined for partition 1 will arrive and complete execution before partition 1 finishes executing any messages originating from itself! The backup trigger is thus avoided, and the LVRT strategy incurs no backups.

In fixed-list strategy, the migrating processor checks the fixed group list (1 4 6 7 8 5 3 2) in order to determine which partition to relocate. Early out-of-work and migrating processors will thus head for partition 1. Partition one's M_{700}^{1-1} will be sent and executed before at least one of the following messages are sent: M_{400}^{2-1}, M_{410}^{3-1}, or M_{430}^{5-1}. Later when they arrive at 1 a backup will result.

In longest task queue strategy the first migrating processor will head for partition 1 because partitions 1, 3, and 5 will all have only a single task (driver) on their respective task queues, and 1 will win out by virtue of its high position on the group list. Once this occurs, the driver of partition 1 will send the first message M_{700}^{1-1} to the first node task in the partition. Thus partition 1 will have two tasks on its task queue which will make partition 1 be the partition of choice for the next migrating processor. In this case longest task queue is seen to work exactly like fixed-list, and fail to a backup for the same reason: M_{700}^{1-1} will be processed too early.

In circular-list strategy, migrating processors will first go to partition 1, and then the group list will circulate to become (4 6 7 8 5 3 2 1). In this way at least one of partition 1's messages to itself will get executed before all of 5 and 3's messages to 1 are finished executing. This is enough to ensure backup.

By changing the group list to (2 5 3 1 4 6 7 8) the importance of its order can be shown. Using this new group list in a fixed-list strategy will avoid backups because partitions 2, 3, and 5 will all go active before 1 and all will be allowed to send their low VRT messages to partition 1 before 1 is allowed to send its high VRT message, M_{700}^{1-1}, to itself.

6.5.2 Group with Longest Task Queue

Figure **6.7** illustrates interactions between partitions in Example 4, where the longest task queue strategy will outperform the fixed-list, circular-list and lowest LVRT repartitioning strategies. Here the experiment's designed-in backup trigger is the processing of M_{700}^{1-1} before M_{280}^{3-1} by partition 1.

In this experiment partitions 4, 6, and 8 are decoupled. Partition 7 is decoupled except for an initial message which it sends to partition 3 at time 1. Partition 1 sends its first message to partition 5. Partitions 4, 6, 7 and 8 each start with three processors. The order of the group list is (7 2 5 1 4 6 8 3).

In the lowest LVRT strategy, partition 5 will be the first partition to be activated by a processor migrating from either 4, 6, 7 or 8 because its LVRT=0. Partition 1 will be the next to be activated because either partition 1 will have a lowest LVRT, or the LVRTs of 1 and 3 will be equal in which case partition 1 will be selected due to its higher position on the group list. Since partition 1 activates before 3 does, backup will soon result.

In the fixed-list strategy, processors migrating from either 4, 6, 7 or 8 will first seek partition 2 because it resides so high on the group list (7 2 5 1 4 6 8 3). In any case the crux is that partition 1 lies higher on the group list than does 3, and its activation prior to 3 will result in a backup.

The circular-list method will also back up. Even though partition 3 will be activated soon after partition 1, the fact that partition 3 has a large number of messages to process will guarantee that partition 1 will have executed its backup-inducing M_{700}^{1-1} before partition 1 receives its M_{280}^{3-1}.

The longest task queue strategy wins out in this experiment because partition 7,

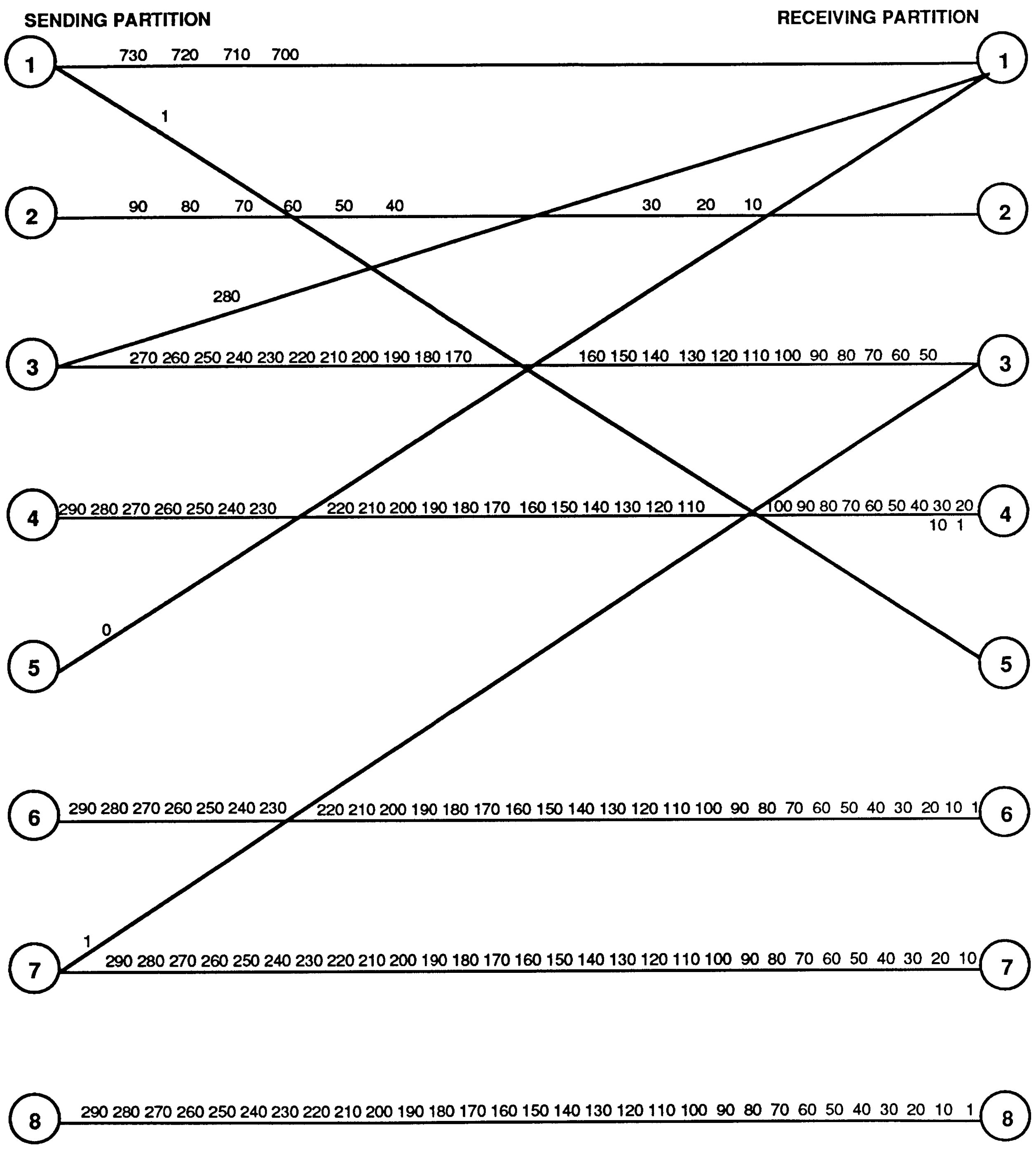


Figure 6.7 Longest TQ Superior over Other Schemes

first on the group list, will receive its processor allocation first. Thus, it will speedily send a message to partition 3, so that 3 will be seen to have two tasks on its task queue, and will therefore get the first migrating processor. Once partition 3 goes active with a single processor, it will then attract additional processors because it has started sending messages to itself, thus lengthening its own task queue. In this way partition 1 processes M_{280}^{3-1} before M_{700}^{1-1} and backup is avoided.

Change the group list order to (2 5 1 4 6 7 8 3) and partition 3 would not receive a message soon enough to prevent partition 2 from receiving the first migrating processor under the longest task queue strategy. Once 2 received this first relocated processor it would then use him to send himself his own messages and thus to lengthen his own task queue. Partition 3 would thus be shut out, and partition 1 would process the backup inducing M_{700}^{1-1} before 3 got out his M_{280}^{3-1}, and a backup would result.

6.5.3 Fixed Group List

Figure **6.8** shows the interactions between partitions in Example 5 where the fixed-list strategy will win out over the circular-list, longest task queue, and lowest LVRT strategies. In this experiment there are two backup triggers. First is partition 1's processing of a M_{700}^{1-1} before M_{500}^{3-1}. The second backup inducer is M_{900}^{8-8} processing prior to M_{10}^{6-8}.

Here partition 2 is decoupled, 4, 6, and 7 are almost decoupled, 5 sends no messages, but receives a message from partition 1 at time 1. Partition 4 begins with five physical processors, partition 5 with one, and partition 7 starts with seven processors. The order of the group list is (3 2 5 6 7 8 4 1). The experiment is split effectively into two parts. In the top half, consisting of partitions 1, 2, 3, and 5, the fixed-list strategy will beat the circular-list and lowest LVRT strategies. In the bottom half (partitions 4, 6, 7 and 8) fixed-list will win against longest task queue.

In lowest LVRT strategy, partition 1 will be activated first by a processor migrating from partition 4, 5, or 7 because it has an LVRT=1. Partitions 2 and 6 will go active before 3 does because they both have an LVRT=10 while 3's LVRT=20. In this case partition 1 will process its M_{700}^{1-1} before M_{500}^{3-1}, and backup will result.

Likewise, circular-list will back up. Partition 3 has twelve messages to process

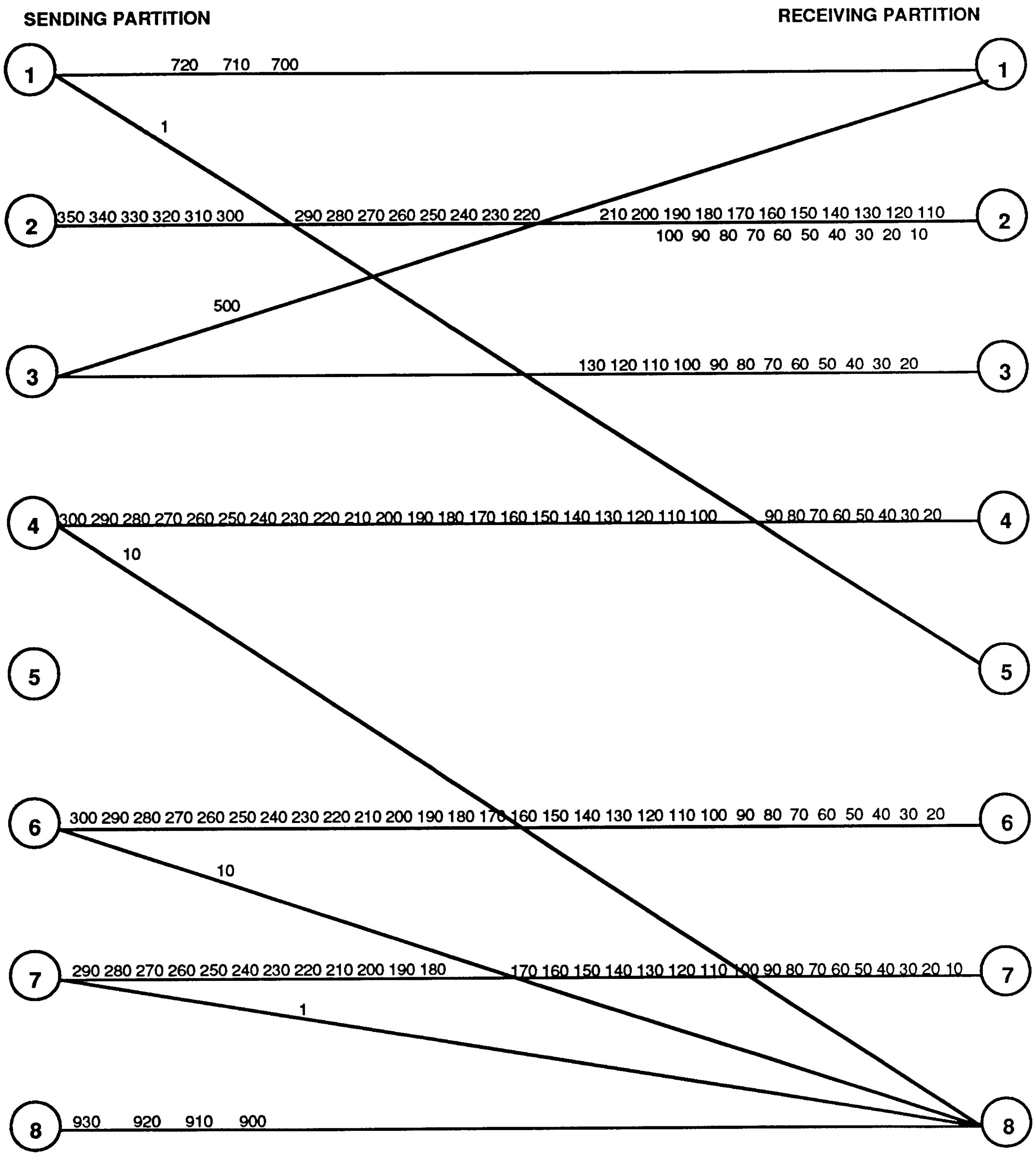


Figure 6.8 Fixed-List Superior over Other Schemes

before it gets out M_{500}^{3-1}. Because the group list is circulated, partition 1 will obtain a processor and process two messages, including M_{700}^{1-1} before 3 sends M_{500}^{3-1}. Backup is the result.

In the longest task queue strategy, partition 8 will back up by processing M_{900}^{8-8} before partition 6 sends M_{10}^{6-8}. The reason 8 gets off to such a fast start is because active partitions 4 and 7 immediately send messages at the simulation's start to 8. The longest task queue strategy then sees three tasks on the 8's task queue and hence sends the first migrating processor to 8.

The fixed-list strategy is an easy winner in the experiment's bottom half (versus longest task queue) since partition 6 is always activated before partition 8 because of its being higher on the group list (3 2 5 6 7 8 4 1). Fixed-list also wins in the top half (versus circular-list and lowest LVRT) because partition 3 executes all of its messages well before 1 executes any.

6.5.4 Circular Group List

Figure **6.9** illustrates the interactions between partitions in Example 6, where the circular-list strategy will outperform fixed-list, longest task queue, and lowest LVRT dynamic repartitioning. Here the experiment again has two backup triggers: partition 1 processes M_{300}^{3-1} before M_{200}^{2-1} and partition 8 processes M_{610}^{8-8} before M_{400}^{6-8}.

In this experiment only partition 7 is decoupled. Partition 6 is almost decoupled. 4 and 5 send no messages, and partition 4 receives a single message from partition 8 at VRT=0. Partitions 4 and 7 begin with six processors apiece. Group list order is (1 5 7 6 3 4 8 2). In the top half of the network (partitions 1, 2, 3 and 5) the circular-list strategy will be superior to both the longest task queue and fixed-list strategies. In the bottom half (partitions 4, 6, 7, and 8) circular-list will win against the lowest LVRT strategy.

The lowest LVRT strategy will activate partition 8 first since its first message has a VRT=0. Thus, M_{610}^{8-8} will cause a backup by being processed by partition 8 before M_{400}^{6-8}.

In both the fixed-list and longest task queue strategies, partition 1 will always be activated before partitions 2 and 3 because it occurs earlier in the group list. 3 activates

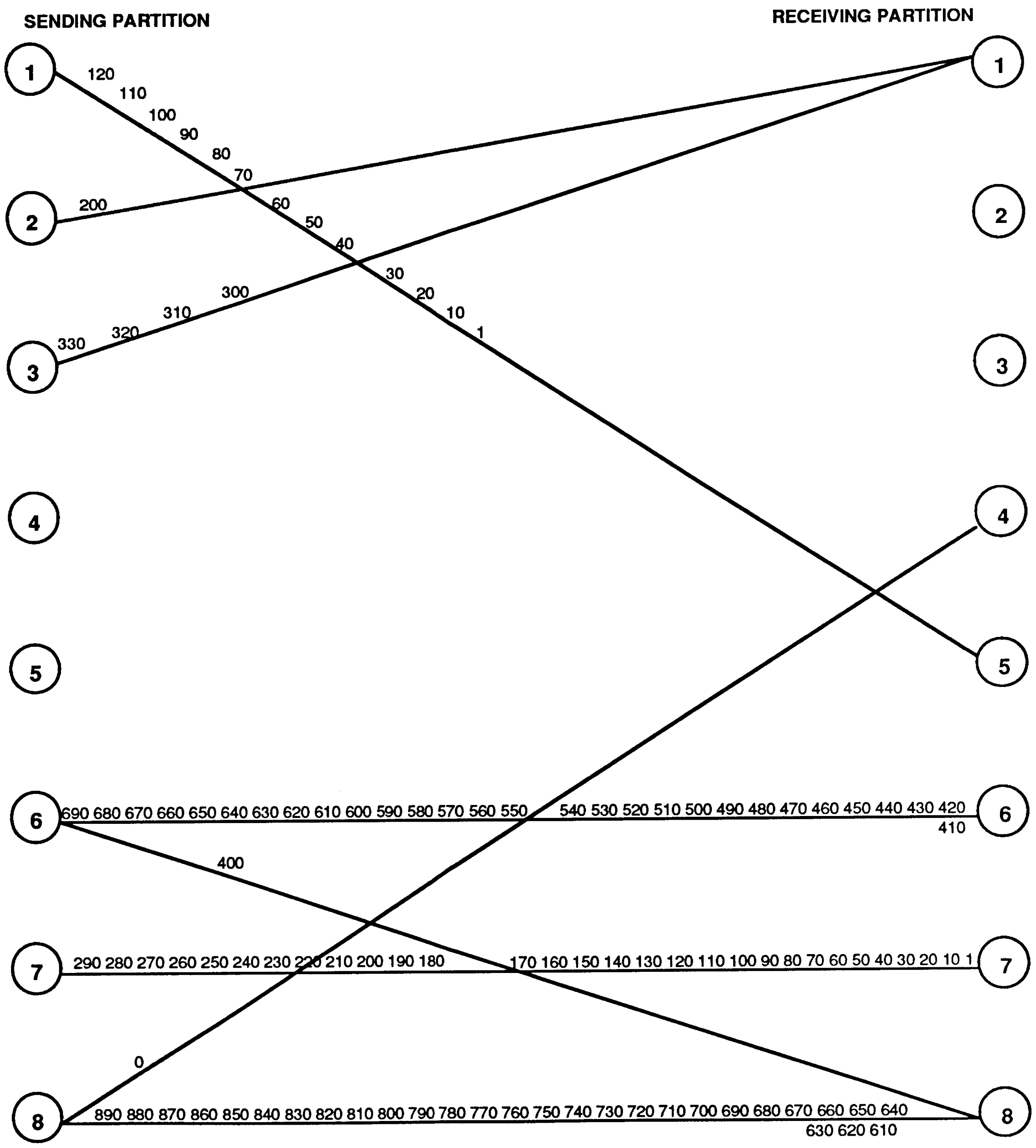


Figure 6.9 Circular-List Superior over Other Schemes

before 2 for the same reason. Because 1 started so early it will be finished processing all of its own messages when it receives its first message from 3, and hence will process

M_{300}^{3-1} prior to M_{200}^{2-1}. Backup will shortly result.

The circular-list strategy will incur no backups, because partitions 4 and 7, being saturated with processors, will fast be freeing processors to the circulating group list (1 5 7 6 3 4 8 2). Partition 1's processing of its own long list of messages to itself will grant partitions 2 and 3 the time they need in order to send their respective M_{200}^{2-1} and M_{300}^{3-1} messages to 1 before it completes the processing of its own intrinsic messages. This is somewhat tricky. The circular-list strategy has temporarily allotted partition 1 only a single processor. This single processor will choose to process the task sending out low VRT message traffic to partition 5, rather than to process the task in its own partition which accepted the high VRT inbound traffic M_{200}^{2-1} and M_{300}^{3-1}. In this way M_{200}^{2-1} and M_{300}^{3-1} will both have arrived on their task's respective message queues before either message is processed. Then when the processor becomes free it will choose to execute the task involving the lower VRT M_{200}^{2-1} versus M_{300}^{3-1}. The messages will thus have been processed in proper order, and backup avoided. In the bottom half of the network, circular-list also works well as partition 6 gets activated before 8 because it occurs earlier in the group list. Partition 6 is easily able to send its first message to 8 before 8 activates, and backup is thus averted.

In Figure **6.10** we summarize the results of the previous four subsections, showing the number of backups for each experiment. These experiments were based on 10 experimental runs each. These results show that each scheduling scheme, under conditions favorable to the scheme in question, was able to execute an example without backups while the other schemes incurred backups.

6.5.5 Estimated LVRT

There are two problems with the EVRT strategy. First, it requires the additional overhead of maintaining a running average of processed messages' LVRTs. Second, the EVRT strategy should not work well in a short simulation, or one that has not yet reached the steady state. The EVRT strategy should be at its best in large simulations well into their steady state, and where message interarrival times can be modeled

	Example 3	Example 4	Example 5	Example 6
Fixed List	3 - 4	3 - 4	0	3 - 4
Circular List	3 - 4	3 - 4	3 - 4	0
Longest TQ	3 - 4	0	3 - 4	3 - 4
Lowest LVRT	0	3 - 4	3 - 4	3 - 4

Number of Backups Based on 10 Runs Each

Figure 6.10 Summary of Extreme Experiments

accurately as independent random variables drawn from the same distribution.

Example 7 contrasts all the deterministic dynamic repartitioning strategies versus the EVRT strategy. Our experiment was run on a 16-node butterfly network. The partitioning method used was horizontal with 16 partitions (discussed in Chapter 5); Figure **5.6** shows an example of the horizontal partitioning method with 16 partitions on a 16-node butterfly network. Of the 16 partitions, each was allocated 2 processors. In this experiment, destinations of each message were random, based upon a uniformly distributed random variable, and average interarrival times of input messages were random, based upon an exponentially distributed random variable. Forty inputs were sent into each of the 16 input nodes with average message interarrival times ranging from 5–180 time units. In this example, the EVRT strategy outperforms the fixed-list, circular-list, longest task queue and lowest LVRT strategies. Figure **6.11** shows the number of backups each scheme had versus the mean interarrival times of messages (MIT). The different schemes that are plotted include: static partitioning (*part*), the fixed-list dynamic repartitioning (*fixed*), the circular-list dynamic repartitioning (*circ*), the longest task queue dynamic repartitioning (*tq*), the lowest LVRT dynamic repar-

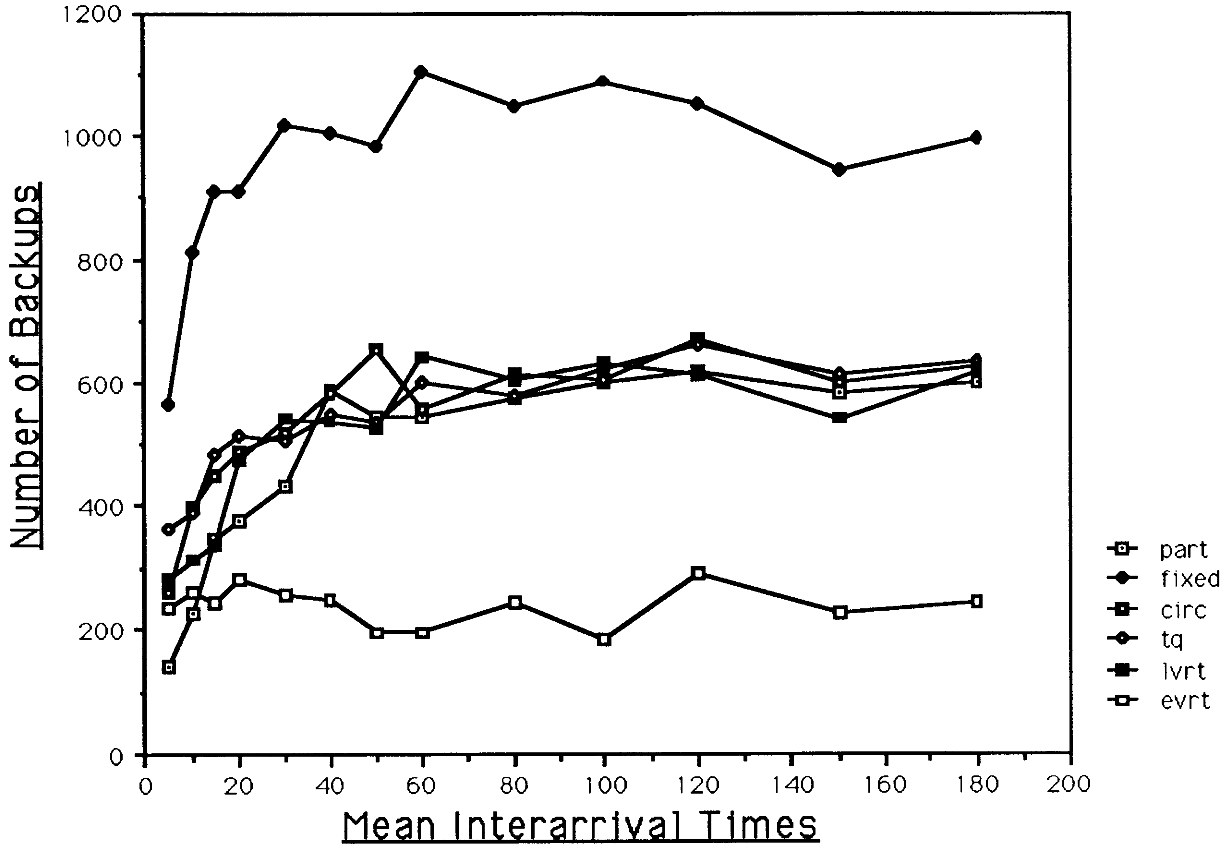


Figure 6.11 Example 7 (Number of Backups versus MIT)

titioning (*lvrt*) and the EVRT dynamic repartitioning (*evrt*). Figure **6.12** shows the processing time versus MIT. The schemes plotted are exactly the same as the previous diagram. The EVRT strategy in some cases experienced less than a third as many backups as did any of the other strategies. By a small margin, the EVRT scheme outperformed all other strategies in terms of processing time for average arrival rates between 30 and 180. These results show that for the EVRT scheme can outperform all other schemes in some cases.

6.6. Continuous Dynamic Repartitioning

Dynamic repartitioning strategies do not continuously relocate processors. If a processor is sent to a partition, then the processor does not leave until there is no work

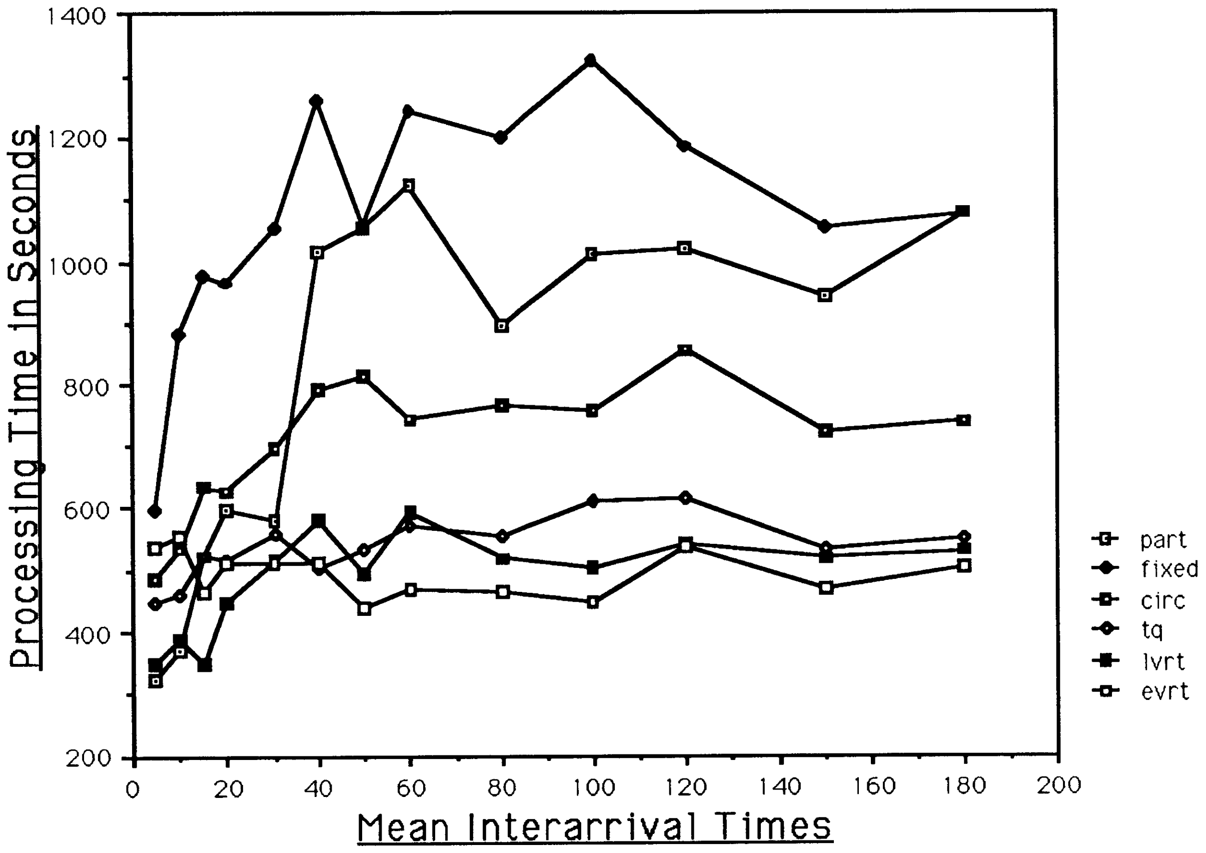


Figure 6.12 Example 7 (Processing Time in seconds versus MIT)

left within that partition. This can create a problem since some partitions can leap far ahead of others in virtual time. Then the partitions that are behind almost inevitably cause the partitions that are ahead to back up. Example 8 contrasts lowest LVRT dynamic repartitioning against lowest LVRT continuous dynamic repartitioning.

In Example 8, we ran continuous dynamic repartitioning on Examples 4, 5, and 6 of the previous section. In two of the three cases no backups arose with the final case having two backups. The results of this experiment are shown in Figure **6.13** along with the processing times for each experiment for both LVRT and the continuous dynamic repartitioning case. Figure **6.13** plots the same categories as Figure **6.4** with the addition of the *backups* measurement which indicates the number of backups the simulation run encountered. The different scheduling policies are either LVRT

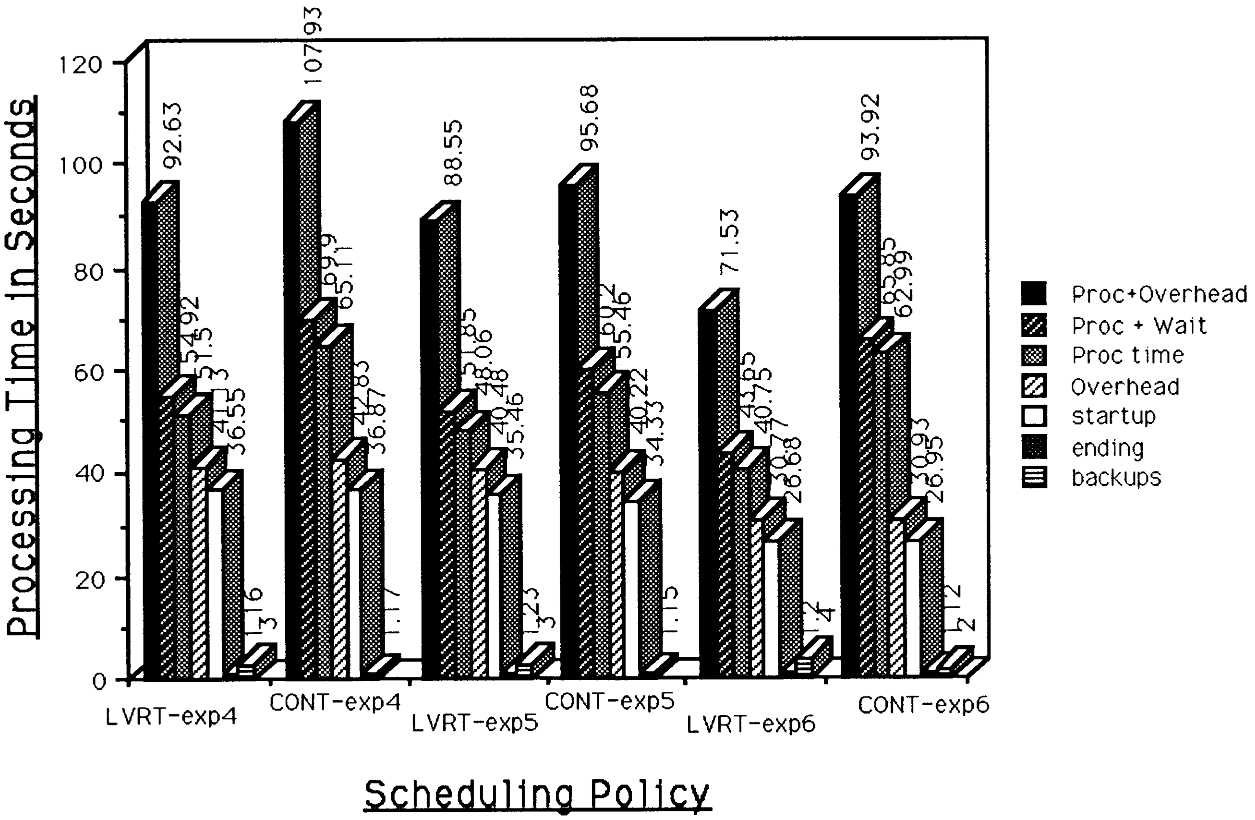


Figure 6.13 Processing times of LVRT versus Continuous Dynamic Repartitioning

or continuous dynamic repartitioning *CONT* performed on the different experiments (examples 4–6) of the previous Section. The example showed a 15 per cent overhead in processing time for continuous dynamic repartitioning.

The reason backups occur in LVRT and not in continuous dynamic repartitioning is that the LVRT algorithm decides upon a processor relocation which may be good for only the short term; on the other hand, in the continuous dynamic algorithm, a processor may relocate after performing only a single task within its present partition. After the processing of each task, a processor will check to see which partition is farthest behind in LVRT and will then relocate there.

One central problem with this lowest LVRT dynamic repartitioning is a 15 per cent additional processing time overhead required in order to invoke the continuous dynamic algorithm. A second problem is that within the continuous dynamic strategy, it is not desirable to lock the group list because that would likely result in a group list bottleneck similar to the non-partitioning scheduling scheme. The whole idea behind partitioning was to avoid the task queue bottleneck by distributing the task queue over the different partitions. By locking the group list, we would just recreate the problem of task queue queue contention in the group list. The following example illustrates a potential problem due to the unlocked group lists: two processors simultaneously looking for work may both relocate to a partition with LVRT=10 and while the partition's next message has a VRT=200. Meanwhile another partition might have an LVRT=30, and should rightfully have gotten the second free processor. However, it did not, and backup will probably soon be the result.

	Example 3	Example 4	Example 5	Example 6
Fixed List	3 - 4	3 - 4	0	3 - 4
Circular List	3 - 4	3 - 4	3 - 4	0
Longest TQ	3 - 4	0	3 - 4	3 - 4
Lowest LVRT	0	3 - 4	3 - 4	3 - 4
Continuous Dynamic	0	0	0	2 - 3

Number of Backups Based on 10 Runs Each

Figure 6.14 Summary with Continuous Dynamic Repartitioning

In Figure **6.14** we summarize the results of the four experiments discussed in Section 6.4 through this section, including continuous dynamic repartitioning. EVRT was not included because it had many more backups due to the shortness of the simulation.

6.7. Summary

This chapter included experiments which showed that any given scheduling strategy can be given simulation conditions where it will win out over all of the other schemes. Prime focus was on the five distinct dynamic repartitioning algorithms, and how they each tried to avoid the processing of a high VRT, backup-inducing message. Given that there is no universally best scheme, each of these schemes is interesting to study because each can potentially perform the best. This observation leaves open the question of which scheme or schemes will excel most frequently in practical situations. In the next chapter, we apply these schemes to more realistic simulations to try to answer this question.

VII. MODEL, RESULTS AND DISCUSSION

7.1. Introduction

This chapter describes a model that characterizes the behavior of our simulation results and illustrates what factors influence the performance. The model is used to study various effects on the simulation system. The different effects that were studied are:

1. The Synchronization Effect
2. The Aggressive Backup Effect
3. The Virtual-Time Delay Effect
4. The Lazy versus Aggressive Message Cancellation Effect
5. The Real-Time Delay Effect
6. The Message Queue Length Effect

For each of these different effects, we develop a theory that uses this model. Once this theory is established we will present the results of our experiments along with a discussion. Finally, we will show the relevance of the effect on other simulation experiments. This model and its experiments will mainly be based on vertical and horizontal partitioning methods because these partitioning methods were easier to analyze.

As a case study we study how these effects affected other partitioning methods and other simulations such as:

1. Random Partitioning on the Network Simulation,
2. Minimum Communications Partitioning on the Network Simulation, and
3. Circuit Simulation.

Along with these case studies we study two minor effects, which are:

1. Partition Saturation, and
2. Importance of Group List Order.

Finally we examine the presence of these effects in the Continuous Dynamic Scheduling scheme.

7.2. Model

We model events (messages) in our system as snowflakes. Each snowflake has a VRT and a Real Receive Time (RRT). The VRT reflects when the snowflake should be processed in relation to other snowflakes. The RRT reflects when the snowflake hits the ground, which corresponds to when the processing task receives the snowflake in real time. Let the ground be a virtual time line where the farther to the right the snowflake lands the larger its VRT. Analogously the processing task is a snowplow which plows up the snowflakes as they land on the ground. The snowplow plows snowflakes from lowest VRT to highest VRTs. The one caveat is that when the snowplow plows over a snowflake there should not be another snowflake that lands behind the snowplow; if this happens, backup will occur. A backup occurs by moving the snowplow backwards on the virtual time line to pick up the stray snowflake, while at the same time dropping snowflakes that were already picked up by the snowplow having VRT greater than the stray snowflake's VRT. Using this model, each snowflake and snowplow have a state consisting of (VRT, RRT).

It is instructive to model a 2-input-node butterfly network, which is much smaller than the 8-input-node or 16-input-node butterfly networks used in actual simulations on Concert. In a 2-input-node butterfly network there are four tasks. Two tasks correspond to the two input nodes while the other two correspond to the two output nodes. We make certain basic assumptions about our model (some of these will change as our discussion builds):

1. Input messages arrive at the two input nodes in increasing VRT order.
2. The destinations of each input-node message is equally likely to go to either of the output nodes.
3. The VRT of each message is random based on an exponential probability distribution (pdf) based on a mean interarrival time (MIT). Each trial of the exponential

pdf is rounded to the nearest whole number. The VRT of the $n + 1^{st}$ message is the VRT of the n^{th} message plus the value returned by a trial of the exponential pdf.

4. Each partition contains exactly one processor.
5. Within each partition the static partitioning scheme is used.
6. Each task has no state. Therefore, there are no conflict delays (see Section 5.4) since conflict delays are implemented using state. In other words, we could make the CD=0.
7. A message's data contains the destination node address.

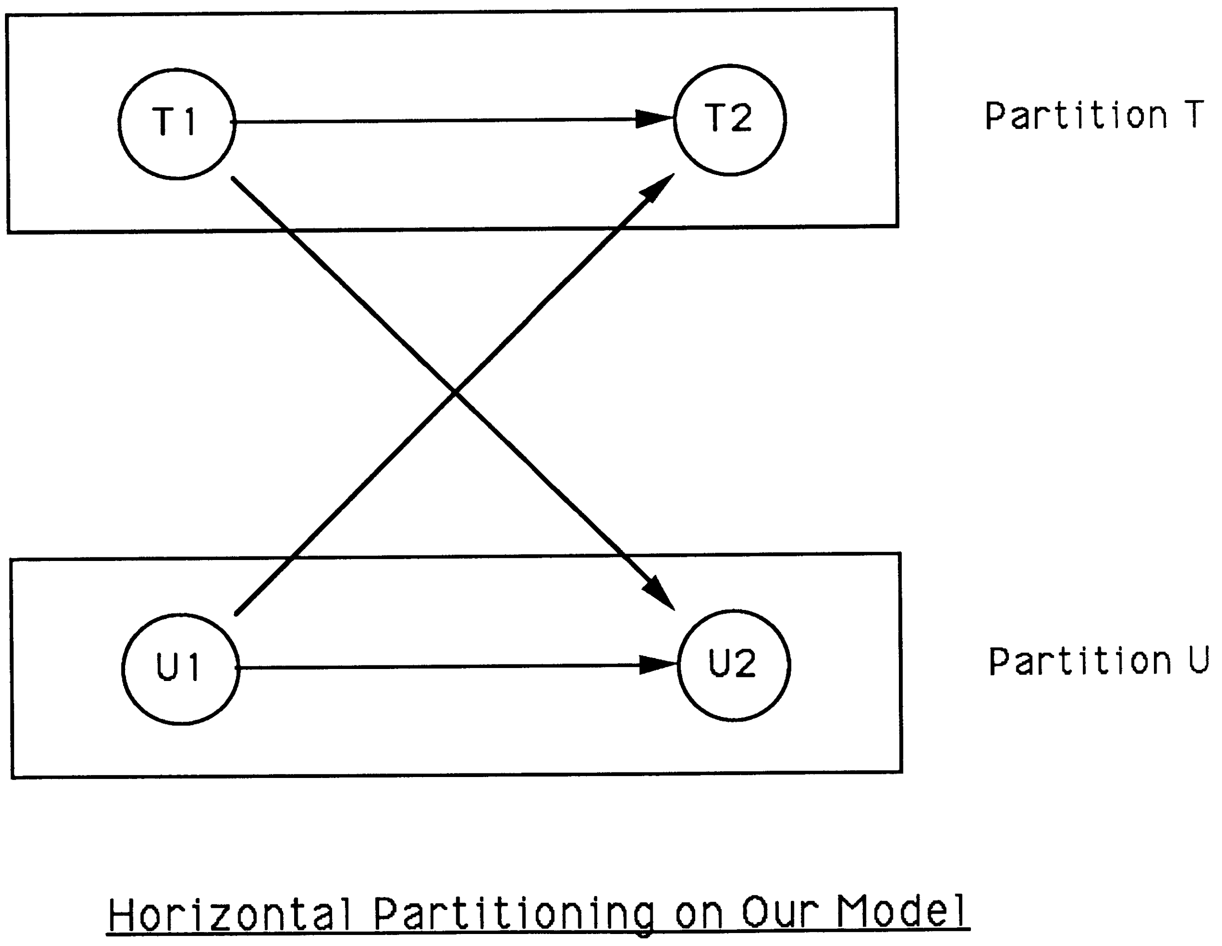


Figure 7.1 Horizontal Partitioning of our Model

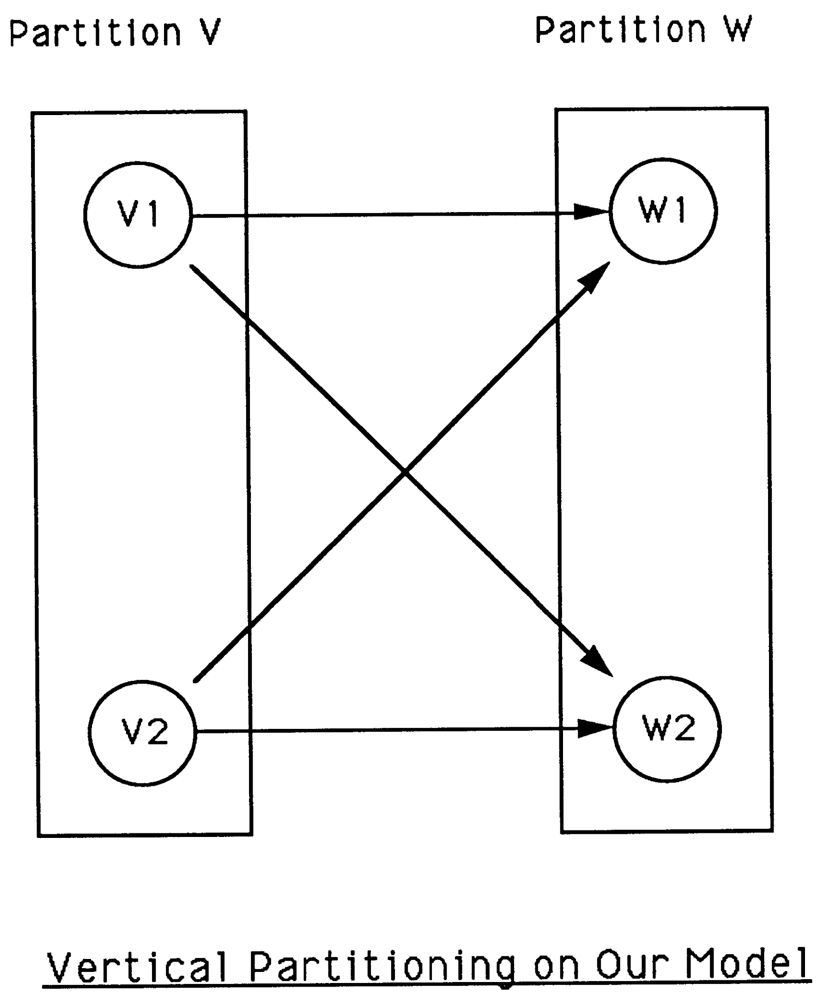


Figure 7.2 Vertical Partitioning of our Model

The way in which we partition the network is called the partitioning method. For this model we study two different partitioning methods: vertical and horizontal. In horizontal partitioning, each partition contains one input task and its horizontally-connected output task, as shown in Figure **7.1**. With vertical partitioning, the first partition contains the two input tasks, while the second partition has the two output tasks, as shown in Figure **7.2**.

We will now describe the various parameters that we can vary in our model as we study the effects these changes make.

1. The mean interarrival times (MIT) between successive messages in virtual time.

2. The nodal delay (ND) for processing a message at any node in virtual time.
3. The inter message delay (IM) between successive messages in real time.
4. The processing time delay (PT) for processing a message at any node in real time.

We shall first study variations in the virtual time parameters while fixing the IM to zero and the PT to one. In our model, the virtual times of successive messages are governed by an exponential probability density function with a mean interarrival time (MIT) of 10 time units.

An input message stream of a task or partition is the set of messages sent to that task or partition preserved in the real-time order in which they were sent. Likewise the output message stream of a task or partition is the set of messages sent from that task or partition with the real-time order preserved. Using our assumptions, the input message stream to an input node is always monotonically increasing in virtual time. Therefore the input stream cannot possibly force a backup in the input nodes.

On the other hand, the input stream to any output node is not always monotonically increasing in virtual time. It is this case that may cause backups. The portion of the input message stream that is not in monotonically increasing order must be sorted into the input message queue before the messages are processed or else backup will occur. For example if the input message stream to a task has VRT of (1, 5, 3, 7), then the task will receive messages with VRTs 1, 5, 3, and 7 in that particular order. Then processing the message with a VRT of 5 (let us call this M_5) before processing M_3 will cause backup. This can occur if the task has not yet received M_3 when it starts to process M_5. However, if M_3 is received before we start processing M_5, it will be sorted onto the input message queue and the messages will be processed in the right order.

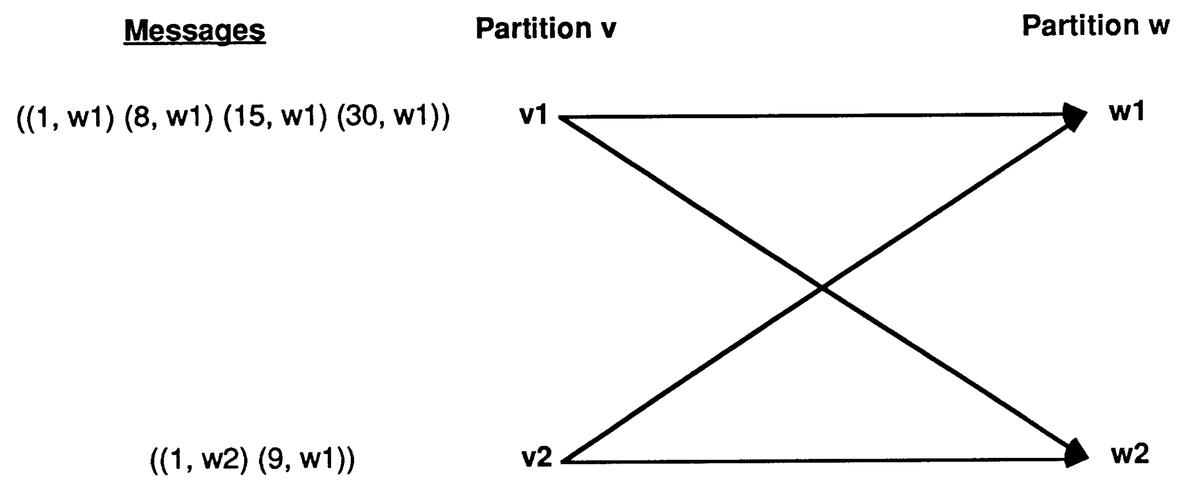
Output message streams of partitions generally supply messages to many tasks' input message streams. Even if the output message stream of a partition is not monotonically increasing, an input message stream that is fed by this output message stream could still be monotonically increasing. This can occur whenever the messages in the output stream that are out of virtual time order go to separate destinations; however, this is unlikely. Thus output message streams that are not monotonically increasing tend to cause backups.

In our network simulation, each message is associated with a destination output node. The destination output node is an element in the data field of the message. When a task receives a message, it forwards a new message with the same destination to the next node on the path to the destination. Each of these messages is distinct; however, in the model and in this thesis we will often refer to these messages as the same message. Although the ND changes the messages' VRTs from node to node, the messages' contents remain the same.

7.3. The Synchronization Effect

The synchronization effect means that the tasks within a partition tend to maintain LVTs that are close to one another. This will occur in any scheduling scheme that uses LVRT to sort the task queues in a partition, as long as none of the tasks' message queues are empty. This has the desirable property that messages sent out on a task's output message stream will be close in virtual time to those of the other tasks within its partition. This is extremely good in vertical partitioning (Figure **7.2**) since both input nodes are in one partition with only one processor. Since there is only one processor and the processor will always process the task with the lowest virtual receive time, the output message stream of this input partition will always be monotonically increasing. Since we assumed the input streams to the input nodes are monotonically increasing in virtual time, no backups will occur in the first partition; likewise, if the second partition receives input streams that are monotonically increasing in virtual time, they too will have no backup.

On the other hand if one uses LVRT $-$ EVRT to sort the task queues in a partition, (as is done in the EVRT scheme) then the tasks will not be synchronized in terms of LVRT but rather in terms of LVRT $-$ EVRT. Hence, it is very likely that the tasks will not have similar LVRTs; therefore, the input partition's output message stream will not be monotonically increasing which will increase the likelihood of backup in the output partition. The problem lies in the fact that our system is based on the quantity LVRT and backups occur whenever the $LVRT < LVT$, which is independent of the quantity, EVRT. Thus the EVRT scheme does not benefit from the synchronization effect.



(20, t2) - A message with VRT=20 and destination task t2

((20, t2) (30, t2)) - An input message queue with two messages

Ouput stream from v using static-LVRT: ((1, w1) (1, w2) (8, w1) (9,w1) (15, w1) (30, w1))

Input stream to w1 using static-LVRT: ((1, w1) (8, w1) (9, w1) (15,w1) (30, w1))

Output stream from v using static-EVRT: ((1, w1) (1, w2) (8, w1) (15, w1) (9, w1) (30, w1))

Input stream to w1 using static-EVRT: ((1, w1) (8, w1) (15, w1) (9,w1) (30, w1))

Synchronization Effect Example

Figure 7.3 Example of the Synchronization Effect

The following example illustrates this problem and is shown in Figure 7.3. In this example we use the following scheduling schemes:

1. static-LVRT which is based on the LVRT to sort the task queues.

2. static-EVRT which is based on the LVRT − EVRT difference to sort the task queues.

In the static-LVRT scheme partition v's processor will process the messages to tasks v_1 and v_2 in LVRT order. Since there is only one processor in partition v the output stream of partition v using the static-LVRT scheme will be monotonically increasing. This is illustrated in Figure **7.3**. On the other hand in the static-EVRT scheme, partition v's processor will process the messages to tasks v_1 and v_2 in LVRT − EVRT order.

When the simulation begins, the LVT and the AT of each task is zero using the EVRT scheme. After processing the $(1, w_1)$ message at v_1, v_1's LVT and AT (see Section **4.9**) are equal to one. Similarly after processing the $(1, w_2)$ message at v_2, v_2's LVT and AT are equal to one. At this time the EVRT (LVT + AT) of both v_1 and v_2 is 2; therefore, the LEVRT = LVRT − EVRT is 6 for v_1 and 7 for v_2. Thus, the message $(8, w_1)$ is processed next at v_1. After processing this message, v_1's AT = 4 and LVT = 8; therefore, v_1's EVRT = 12 and v_2's EVRT = 2. Since v_1's LEVRT = 15 − 12 = 3 and v_2's LEVRT = 7, v's processor will process the message $(15, w_1)$ at v_1. Proceeding in this same fashion the output stream from partition v is shown in Figure **7.3**.

In this example, the output stream of messages from partition v using the static-LVRT scheme is monotonically increasing; on the other hand, using the static-EVRT scheme the output message stream is not. Therefore the static-LVRT scheme creates a synchronization effect whereas the static-EVRT scheme does not. If v's output message stream is not monotonically increasing, the input stream to w_1 will also not be monotonically increasing. Therefore, if the messages arriving at w_1 are processed as quickly as they are received, backup will occur.

The synchronization effect is relevant when we compare vertical and horizontal partitioning. In vertical partitioning using a non-EVRT scheme, each task in a partition will have its LVRT close to that of the other tasks in the partition. If we examine the input partition we can observe that the output is a message stream that is monotonically increasing because its tasks are synchronized and the input stream is monotonically increasing. Thus no backups will occur in the output partition as well as in the input partition. In general, the EVRT scheme does very badly in vertical partitioning because it does not create this effect, whereas the LVRT scheme does.

The synchronization effect can hinder the performance of horizontal partitioning when using the static-LVRT scheme. This is true in our two-partition model of Figure **7.1** as well as in our multiple partition networks H_8 or H_{16}. In horizontal partitioning, the synchronization effect helps to keep the tasks within partitions synchronized with respect to each other in virtual time. Given two partitions, A and B, that communicate with each other, then one partition will typically have a lower LVRT (a partition's LVRT is the LVRT of the task with the lowest LVRT within its partition). Each partition's output message stream is monotonically increasing due to the static-LVRT scheme. If partition A's LVRT is greater than B's and a task with the lowest LVRT in B sends a message to a task in A, then that task in A must back up.

However, these backups cause synchronization *between* partitions. Every time a message sent between partitions A and B causes a backup, the LVRT's of A and B will get closer. However, each partition is still susceptible to further backups if C, the partition with the lowest LVRT, sends a message to either A or B. Furthermore, the messages that are reprocessed by the backup caused by a message between A and B may again need to be reprocessed after C sends a message. Therefore, the benefit of synchronization from backups comes at the great expense of repeated computations due to backups.

7.4. The Aggressive Backup Effect

When a backup occurs at a task, some messages from the *processed message queue* are removed and placed on the *unprocessed message queue.* Reprocessing these messages quickly can cause additional backups. This is called the aggressive backup effect. These messages are reprocessed quickly because an inappropriately high priority is given to

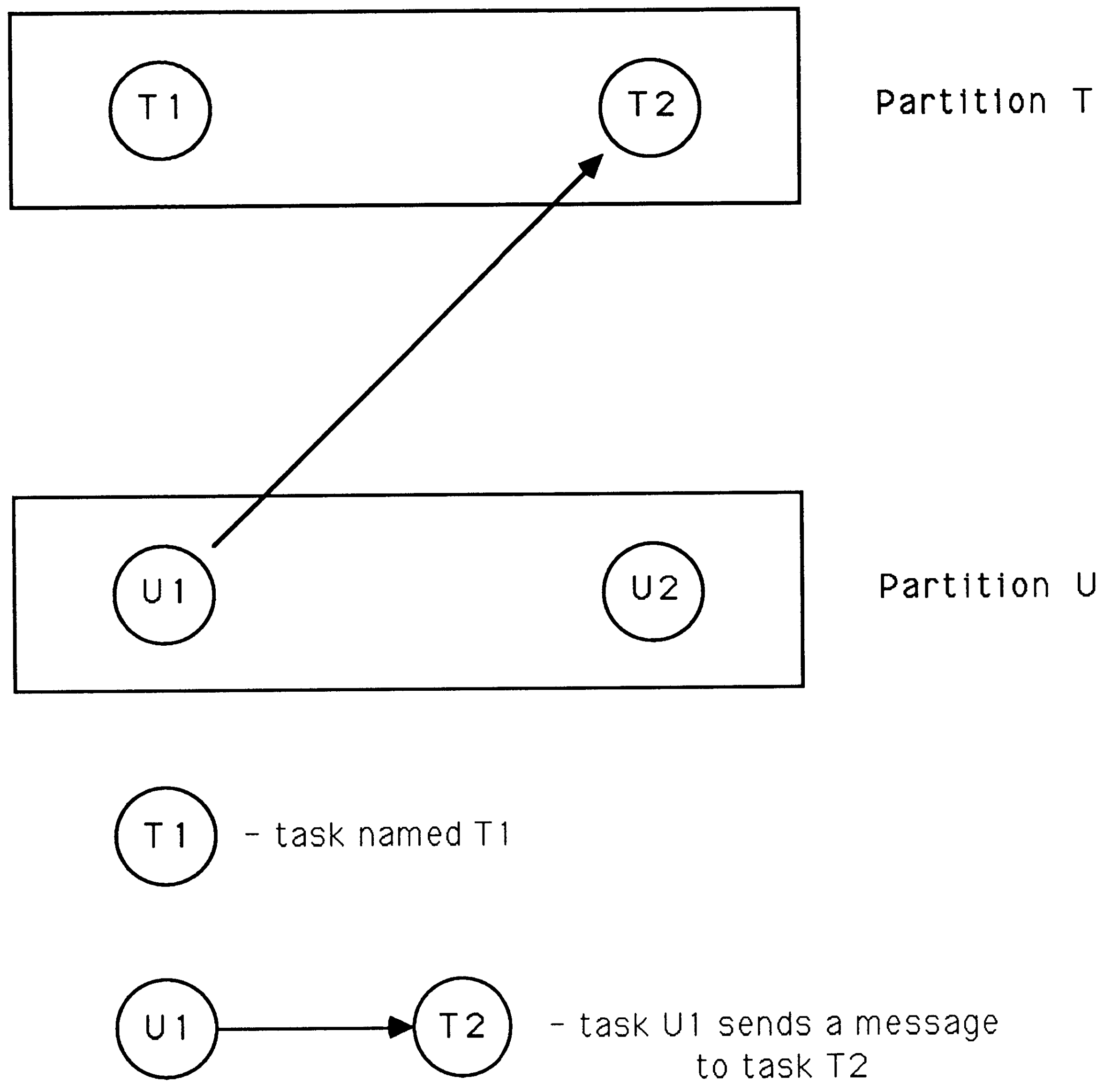


Figure 7.4 Example of the Aggressive Backup Effect

the backed-up task due to its low LVRT. In our simple four-node example, this effect is only present in horizontal partitioning.

The following example illustrates the aggressive backup effect. Consider the four-node network of Figure 7.4. T_1 and T_2 are tasks in the partition T, while U_1 and U_2 are tasks in the partition U. Assume that T_1 and T_2, as well as U_1 and U_2, will have similar LVRTs at the start of the example. Suppose that U_1 sends a message to T_2

with VRT less than T_2's LVT. This causes T_2 to back up. We will go on to explain how this backup could cause even more backups.

When T_2 is backed up, it will be given processing priority in its partition due to its lower LVRT, and will be processed exclusively by T's processor until its LVRT reaches that of T_1. However, partition U will not adjust its processing behavior. U's processor will continue to process the node with the lowest LVRT. Since U_1 and U_2 will have similar LVTs because they are in the same partition, we can expect the processor assigned to this partition to split its processing time between tasks U_1 and U_2. Since over a period of time task T_2 will receive twice as much processing time as U_1, the LVRT of U_1 will eventually fall behind that of T_2. If U_1 then sends another message to T_2, an additional backup will occur. This can happen repeatedly.

We establish a few definitions for this example. Partition T, which is the partition with the backed-up task, is said to be a backed-up partition. Partition U, which contains the task that caused the backup, is called the preempting partition. Task U_1, the task that caused the backup, is called a preempting task.

The problem arises because the preempting task U_1 is given lower priority than the backed-up task T_2 even though they have similar LVTs immediately after backup. This problem can potentially be alleviated by not giving full priority in the backed-up partition to the backed-up task. Specifically, this means having partition T's processor sometimes process T_1. This is safe because messages processed at T_1 destined for T_2 will only be placed at the end of the queue because T_2's input message queue is long and because T_1's LVT is greater than T_2's. Also, messages from T_1 destined for U_2 will be placed at the end of U_2's input message queue because T_1 has a higher LVT than U_1 and U_2. T_1's LVT is the highest because before backup, T's LVRT must be higher than U's or else the backup would not occur. Once T_2 backs up it will have a higher LVT than T_1 also. Since the messages are placed at the end of the queues, the chance of backup is slim.

Therefore, the LVRT strategy is not the best strategy to use for horizontal partitioning because it gives priority to the task that was backed up. In order to balance the processing power between both tasks in the backed-up partition, one may try the EVRT scheme. In fact the EVRT scheme, which does not give extra priority to the backed-up task, is less susceptible to this aggressive backup effect.

7.5. Virtual-Time Delay

7.5.1 Theory

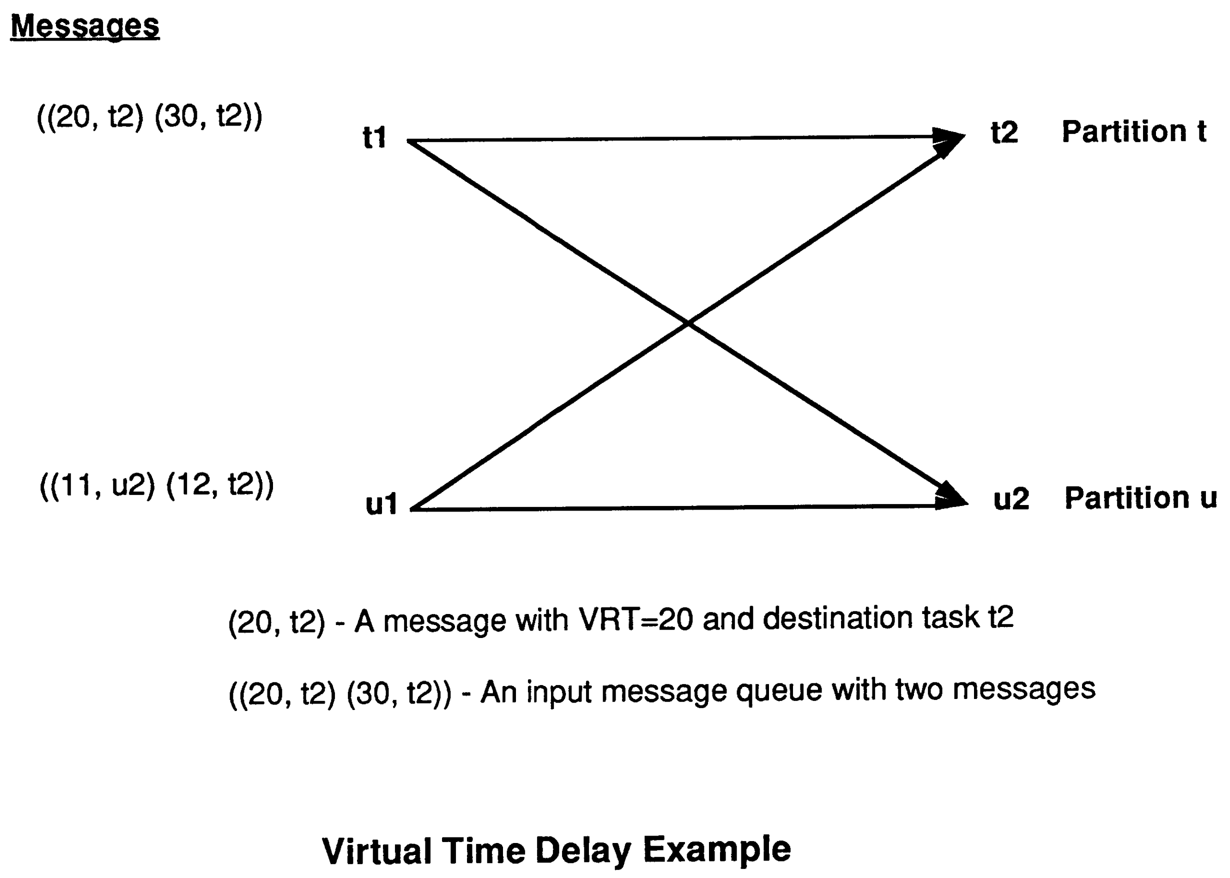

Messages
((20, t2) (30, t2))
t1
t2 Partition t
((11, u2) (12, t2))
u1
u2 Partition u
(20, t2) - A message with VRT=20 and destination task t2
((20, t2) (30, t2)) - An input message queue with two messages
Virtual Time Delay Example

Figure 7.5 Example of the Virtual-Time Delay Effect

Here we shall describe how mean interarrival times (MIT) and nodal delay (ND) (defined in Chapter 5), can affect the performance of our simulation. The performance of the simulation is dependent on ratio of MIT:ND. We continue with our model simulation of a four-node butterfly network. As we vary the ratio MIT:ND the performance of the simulation changes. The virtual time delay between successive messages affects horizontal partitioning much more than vertical partitioning. Let us look at an example of horizontal partitioning with non-zero MIT and ND=0. Consider our model with four tasks labeled t_1, t_2, u_1, and u_2 as shown in Figure **7.5**. In this example, the notation (20, t_2) represents a message with VRT = 20 and destination task t_2. The notation ((20,t_2) (30,t_2)) represents an input message queue with two messages.

If the input messages to t_1 have virtual receive times and destinations as ((20,t_2) (30,t_2)) and messages to u_1 have times and destinations as ((11,u_2) (12,t_2)), then backup will occur. The processor in partition t will first process the message (20,t_2) at t_1 while the processor in partition u will process the message (11,u_2) at u_1. Then since the the nodal delay is 0, t's processor will process the message (20,t_2) at t_2 while u's processor will process the message (11,u_2) at u_2. Next t's processor will process message (30,t_2) at t_1 and u's processor will process message (12,t_2) at u_1. Processing message (12,t_2) will cause t_2 to back up because the message has a VRT of 12 and task t_2 will have a LVT of 20.

Now consider the case when ND=11. After processing the messages (20,t_2) at t_1 and (11,u_2) at u_1, both processors will choose to process the second messages at t_1 and u_1 respectively. This is because t's processor has to choose between processing message (30,t_2) at t_1 or (31,t_2) at t_2, while u's processor has to choose between processing message (12,t_2) at u_1 and (22,u_2) at u_2. Since the messages at t_1 and u_1 have lower VRT they are processed first, thus preventing backup. Therefore changing the nodal delay can affect the simulation.

Changing the MIT will have a similar affect for this example: it is only the ratio of MIT to ND that affects the performance because the results of the example will not change if both MIT and ND are changed by the same factor.

In vertical partitioning, varying the MIT or the nodal delay has no effect because all of the messages to all the tasks within a partition are affected equally. For example if we increase the ND while using vertical partitioning, then the input partition (the partition with all the input nodes) is unaffected because none of its input messages' VRTs have changed. In the output partition all of the input messages have their VRTs increased by ND. Since this ND is added to each message, the simulation runs exactly as before, processing the same messages at the same time. The only difference is each output message has an additional $(2 \cdot ND)$ added to its VRT. Likewise, increasing the MIT will simply change the virtual time delays between successive messages by the same amount. Since all messages to the input partition are affected equally, this will not change the simulation run except to add virtual time delays between successive output messages' VRTs; likewise, varying the MITs will not affect the simulation run at the output partitions.

If horizontal partitioning is used, varying the ND or MIT does affect the performance. For example if we increased the ND, then messages to t_2 and u_2 have their virtual times increased by the ND. This effect was shown in the example of Figure 7.5, and decreased the number of backups. Thus by increasing the ND in horizontal partitioning we affect the tasks in a partition differently. Increasing the ND gives priority to the input-node tasks, which is good in our model because it makes our partitioning method seem more like vertical partitioning. In fact, if the ND is very large, such that the processors process all of the messages in the input-nodes first before processing any messages for the output nodes, then no backups will occur. Likewise, decreasing the MIT will also decrease backups due to the uneven way it affects the tasks in a horizontal partition. Therefore the number of backups in horizontal partitioning can vary due to changes in MIT and ND.

7.5.2 Experiments

This section describes the results of large-scale experiments run on Concert that confirm our theories about the following:

1. Synchronization Effect
2. Aggressive Backup Effect

3. Virtual-Time Delays

7.5.2.1 H_8-Partition Case

In this section we discuss the results of running the 16-input-node butterfly network with the H_8 partitioning while we varied the MIT between 5 and 180 with ND equal to 10. This partitioning method is shown in Figure **5.5**. Each partition was assigned four processors, since each partition had equal amounts of work.

Horizontal Partitioning with 8 Partitions

Initial Processor Allocation	(4 4 4 4 4 4 4 4)
Effects of Increasing MIT on # of Backups	# of Backups increases for all schemes except for EVRT where it remains constant
Scheme Ordering Based on # of Backups (ordering from worst to best)	1) Fixed-list 2) Partitioning, Circular-list, TQ and LVRT 3) EVRT (except static partitioning for low MIT)
Effects of Increasing MIT on Processing Time	Processing time increases for all schemes except for EVRT where it remains constant
Scheme Ordering Based on Processing Time (ordering from worst to best)	1) EVRT, Fixed-list, and Partitioning 2) Circular-list 3) TQ 4) LVRT

Figure 7.6 Summary of Horizontal Partitioning with Eight Partitions

A summary of the description and results of this experiment can be found in Figure **7.6**. The results of this experiment are shown in Figures **7.7** and **7.8**. Figure **7.7** shows the number of backups for each of the scheduling policies as the MIT for messages increases. The policies are static partitioning (part), fixed list (fixed), circular list (circ), longest task queue (tq), lowest LVRT (lvrt), and lowest EVRT (evrt). As the MIT increases, backups increase in all the different scheduling policies, except for the EVRT strategy, where backups remain constant. The EVRT strategy has the fewest

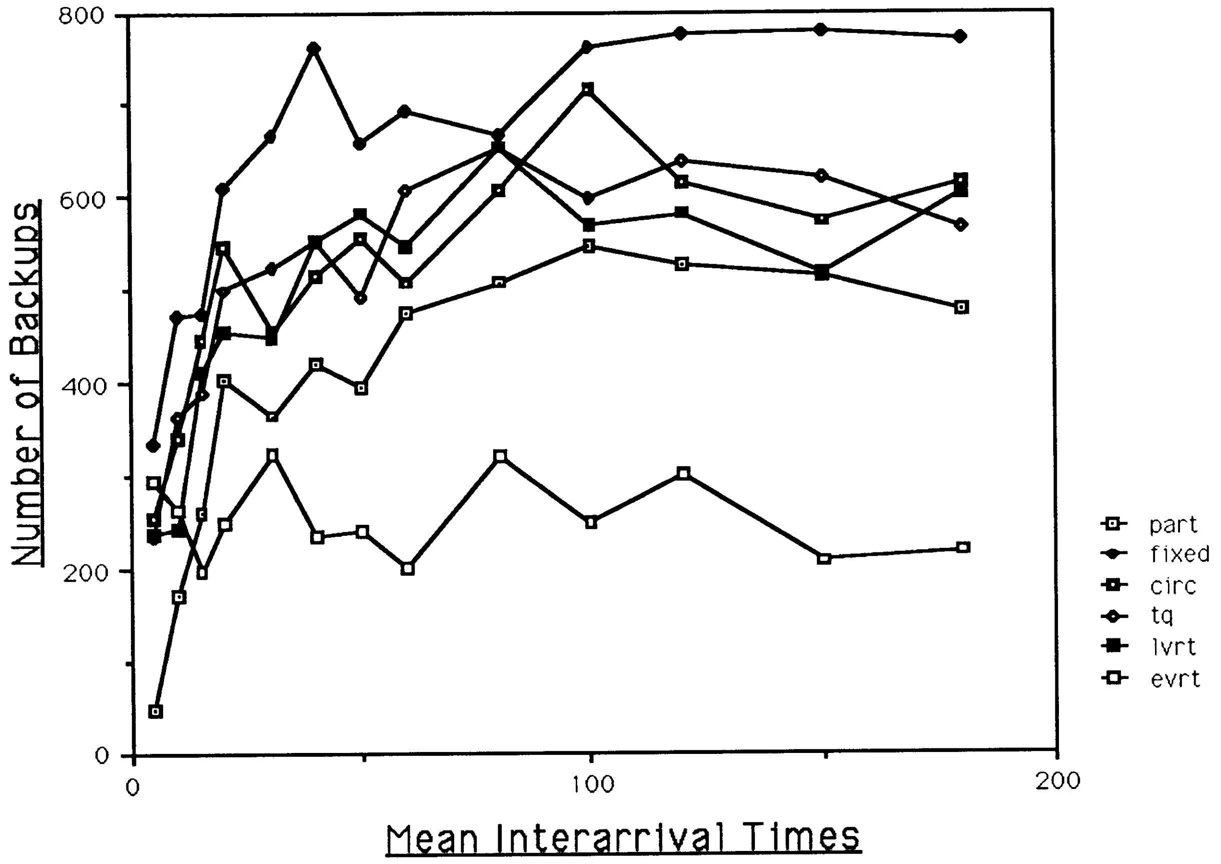


Figure 7.7 Number of Backups versus MIT for H_8

number of backups for MIT between 15 and 180, whereas the fixed-list strategy had the most backups.

Figure 7.8 shows the processing times (in seconds) for each of the scheduling policies as the MIT increases. As the MIT increases, processing times increase in all the different scheduling policies, except for the EVRT strategy, where it remains constant. The LVRT had the fastest processing times. Note that the EVRT scheme did not have the lowest processing time even though it had the fewest backups. We attribute this to additional processing overhead to compute the ATs and the EVRTs in the EVRT scheme.

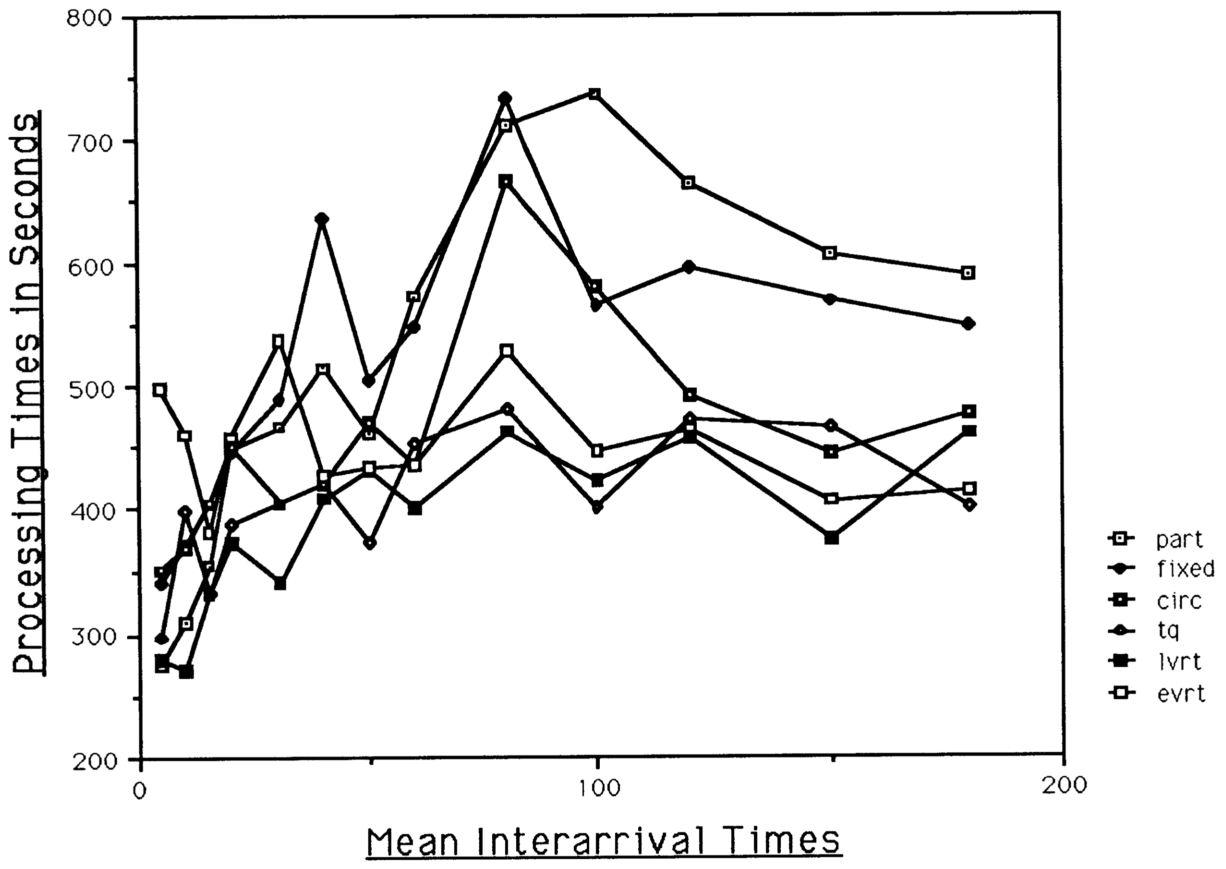


Figure 7.8 Processing Time versus MIT for H_8

7.5.2.2 H_{16}-Partition Case

In this section we discuss the results of running the 16-input-node butterfly network with the H_{16} partitioning while we varied the MIT between 5 and 180 with ND equal to 10. This partitioning method is shown in Figure **5.6**. Each partition was assigned two processors, since each partition had equal amounts of work.

This experiment is summarized in Figure **7.11** in the same manner as the previous experiment. The results of this experiment are shown in Figures **7.9** and **7.10**. Figure **7.9** shows the number of backups as the MIT increases. As the MIT increases, backups increase in all of the different scheduling policies, except for the EVRT strategy, which remains constant. The EVRT strategy has the fewest number of backups for MIT between 15 and 180, whereas the fixed-list scheme had the most backups. Note that

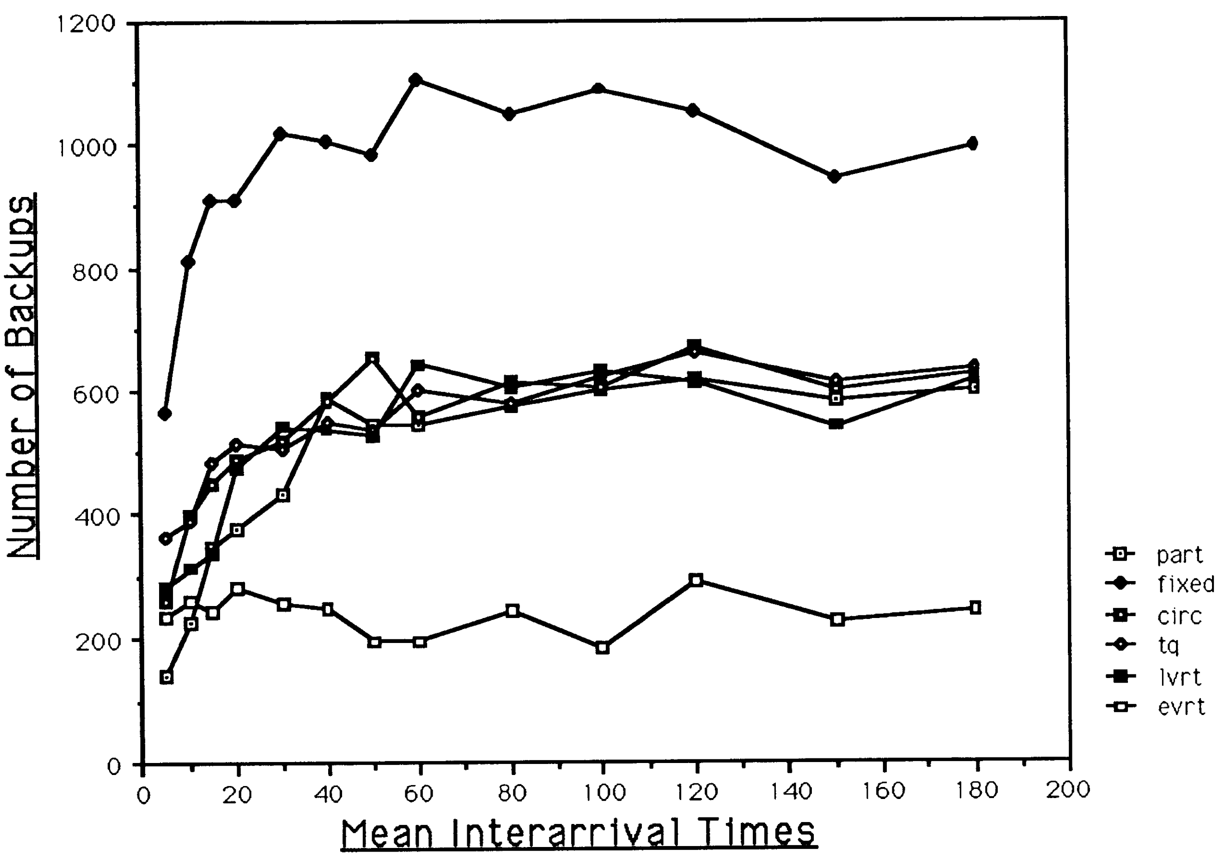


Figure 7.9 Number of Backups versus MIT for H_{16}

the trend of each scheduling scheme is clearer in the H_{16} case than the H_8 case.

Figure **7.10** shows the processing times of the different scheduling schemes as the MIT increases. As the MIT increases, processing times increase for all the scheduling policies, except for the EVRT scheme, which remains constant. The EVRT strategy has the lowest processing times when MIT is greater than 40. The fixed-list scheme had the worst times. Note that the EVRT scheme continues to improve in relation to the other schemes as the MIT increases; for low MIT it was one of the worst schemes, but for higher values of MIT it was the best scheme.

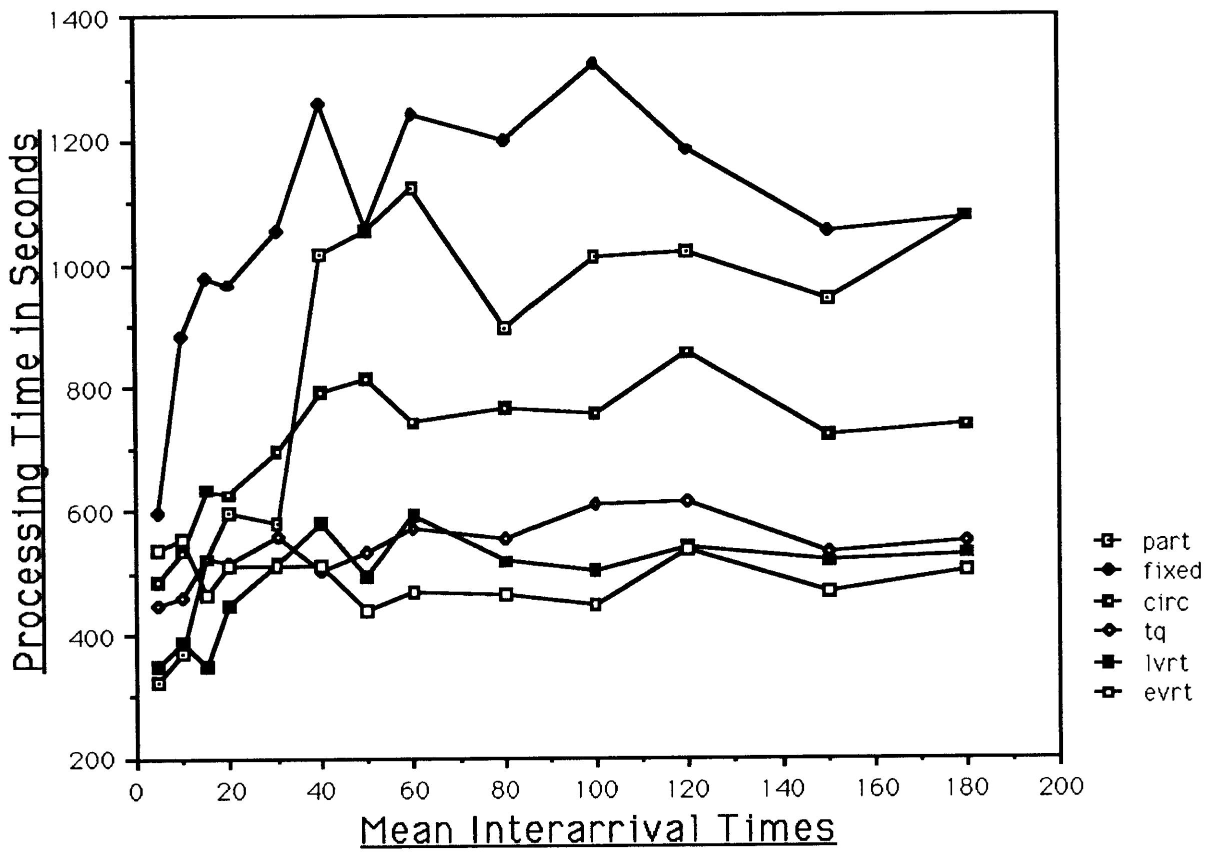


Figure 7.10 Processing Time versus MIT for H_{16}

7.5.2.3 V_6-Partition Case

In this section we discuss the results of running the 16-input-node butterfly network with the V_6 partitioning while we varied the MIT between 5 and 180 with ND equal to 10. This partitioning method is shown in Figure **5.7**. Except for the static partitioning policy, the first partition started with all 32 processors since it was the one with all of the work at the beginning of the simulation; this was optimal. For static partitioning the first two partitions started with six processors and the last four partitions started with five processors; this was not optimal, but since each partition had about the same amount of work it was close to optimal given that processors do not move. The results of this experiment are shown in Figures **7.12** and **7.13**. Figure **7.12** shows the number of backups as the MIT increases. Varying the MIT does not affect the results. The

Horizontal Partitioning with 16 Partitions

Initial Processor Allocation	(2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2)
Effects of Increasing MIT on # of Backups	# of Backups increases for all schemes except for EVRT where it remains constant
Scheme Ordering Based on # of Backups (ordering from worst to best)	1) Fixed-list 2) Partitioning, Circular-list, TQ and LVRT 3) EVRT (except static partitioning for low MIT)
Effects of Increasing MIT on Processing Time	Processing time increases for all schemes except for EVRT where it remains constant
Scheme Ordering Based on Processing Time (ordering from worst to best)	1) Fixed-list 2) Partitioning 3) Circular-list 4) TQ and LVRT 5) EVRT

Figure 7.11 Summary of Horizontal Partitioning with 16 Partitions

LVRT scheme was the best while the EVRT was the worst in terms of backups.

Figure **7.13** shows the processing times of the different scheduling schemes as the MIT increases. The MIT has no systematic effect on processing times for this partitioning method. The LVRT scheme was the best for the majority of MITs. The EVRT scheme was the worst for the majority of MITs. The results are summarized in Figure **7.14**.

7.5.3 Discussion

7.5.3.1 Horizontal Partitioning

These experimental results showed the the following effects on the horizontal partitioning method:

1. The Synchronization Effect
2. The Aggressive Backup Effect
3. The Virtual-Time Delay Effect

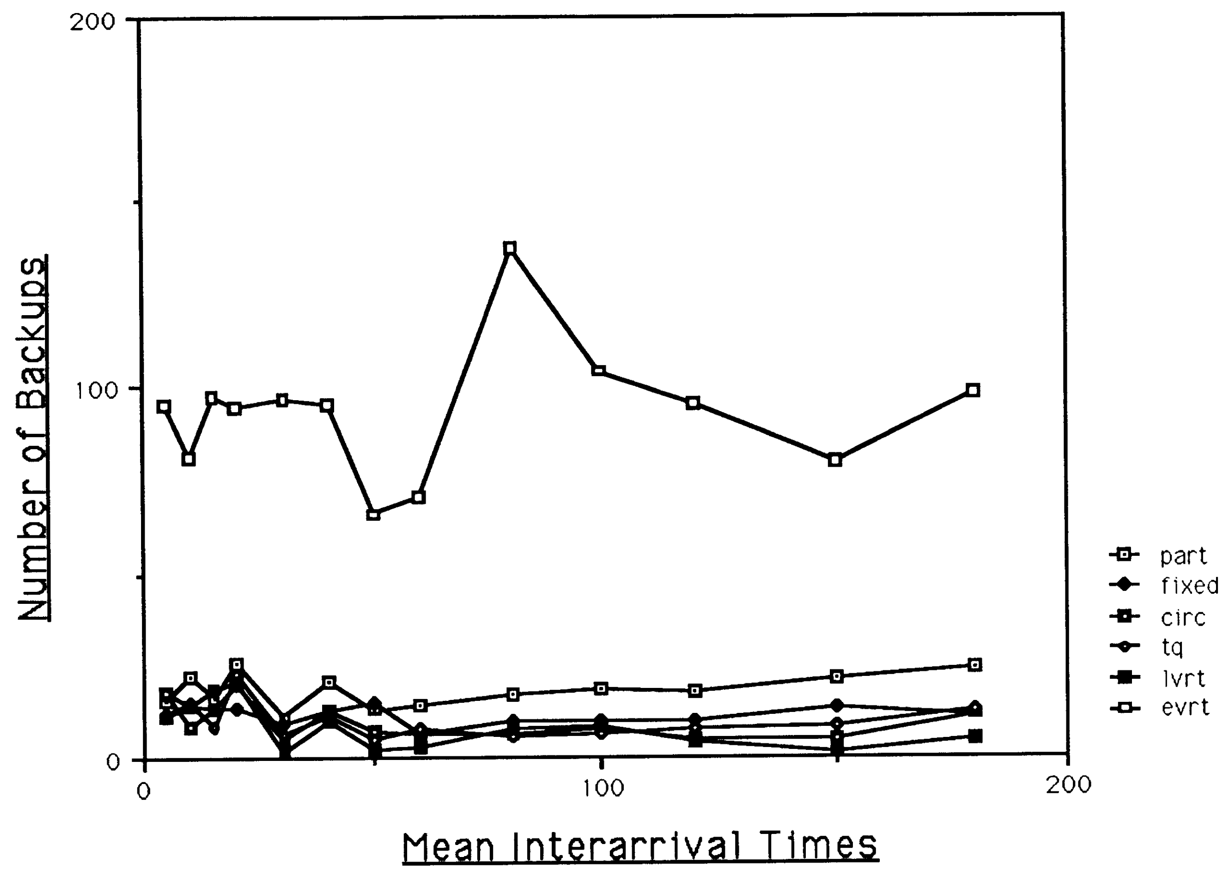


Figure 7.12 Number of Backups versus MIT for V_6

Summarizing the results:

1. The fixed-list scheme performed the worst
2. The EVRT scheme performed the best
3. Horizontal partitioning performed poorly compared to vertical partitioning

The fixed-list scheme is generally bad for horizontal partitioning because it will concentrate processors in only one partition (the first partition on the list that has work available). This allows one partition to get far ahead of other partitions which should have equal priority, making the network even less synchronized in terms of LVT. Thus when other partitions send messages to this far-ahead partition, they will inevitably cause backups. By comparing Figures **7.7** (eight partitions) and **7.9** (16 partitions), it can be seen that the number of backups produced by the fixed-list scheme increases

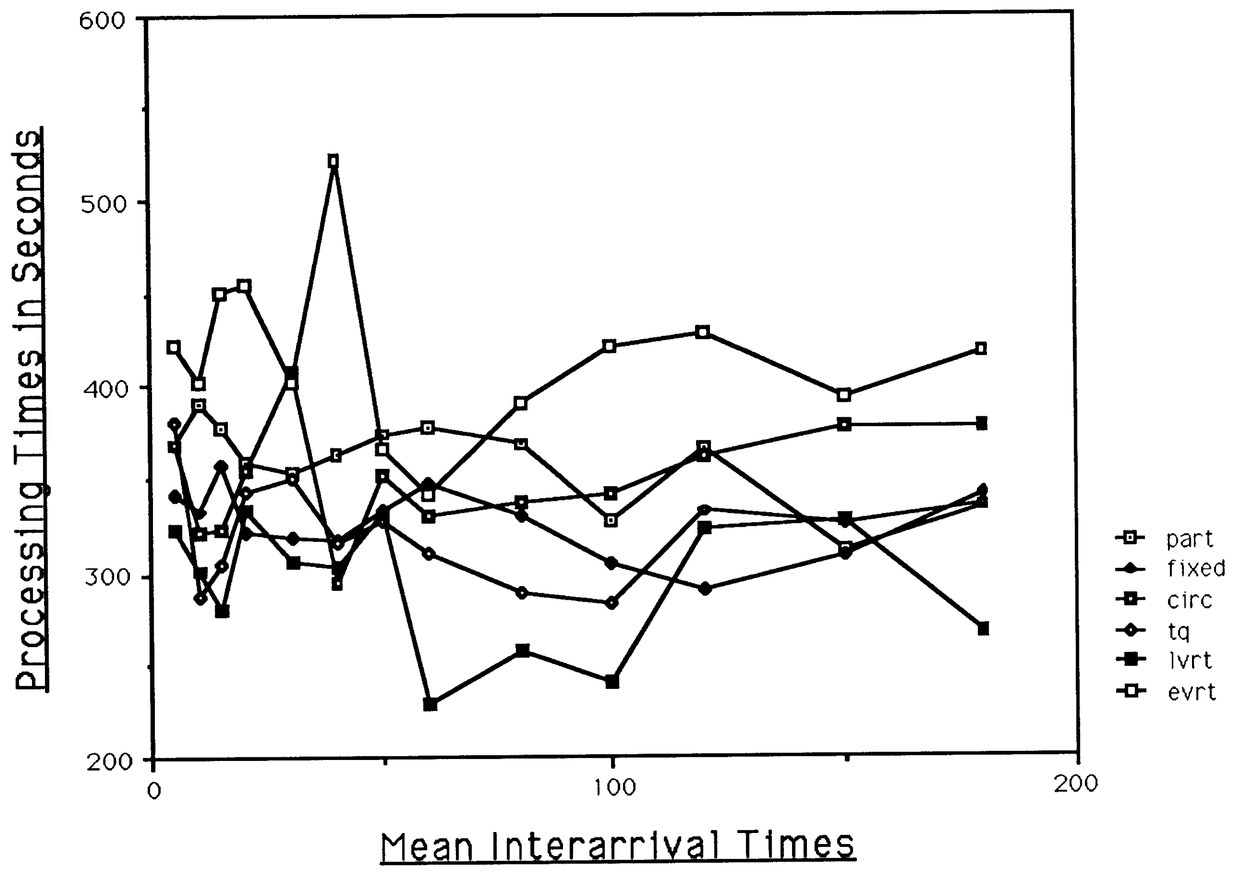


Figure 7.13 Processing Time versus MIT for V_6

when more partitions are used. By comparing Figures **7.8** (eight partitions) and **7.10** (16 partitions), it can be seen that the fixed-list scheme had longer processing time when more partitions are used. Our data show that the fixed-list scheme is better in H_8 than in H_{16}. We attribute this to the synchronization within partitions in H_8. We have stated that in horizontal partitioning, it is very likely for a message sent between two tasks in different partitions to cause backups. If these two tasks were in the same partition, then it would be less likely for backups to occur because the tasks would be synchronized. Therefore, decreasing the number of partitions from 16 to 8 decreases the number of inter-partition links, which reduces the number of backups. This is generally true for the other scheduling schemes but not to such a large extent.

In comparing the number of backups and processing times of the horizontal par-

Vertical Partitioning with 6 Partitions

Initial Processor Allocation	(32 0 0 0 0 0) (except static) (6 6 5 5 5 5) (static)
Effects of Increasing MIT on # of Backups	None
Scheme Ordering Based on # of Backups (ordering from worst to best)	1) EVRT 2) Partitioning 3) Fixed-list, Circular-list, and TQ 4) LVRT

Effects of Increasing MIT on Processing Time	None
Scheme Ordering Based on Processing Time (ordering from worst to best)	1) EVRT 2) Partitioning and Circular-list 3) Fixed-list, TQ and LVRT

Figure 7.14 Summary of Vertical Partitioning with 6 Partitions

titioning method to those of vertical partitioning, the horizontal partitioning method suffers from more backups and longer processing times. We attribute this to the aggressive backup effect which, as discussed in Section 7.4, can affect simulations using horizontal partitioning.

The EVRT scheme performs best in terms of backups and processing times as shown in Figures 7.7, 7.9 and 7.10. The EVRT scheme is the best because it does not sort the tasks in terms of LVRT but rather (LVRT − EVRT), which does not exclusively process the task with the lowest LVRT, but instead allows processing of other tasks with higher LVRT. Thus the EVRT scheme does not give priority to the backed-up task, and does not suffer from the aggressive backup effect.

The horizontal partitioning method also fared poorly due to increasing virtual time delays (increasing MITs) where as the MITs increased each scheduling scheme except EVRT performed worse. Each deterministic scheduling scheme had increasing numbers of backups and longer processing times as the MITs were increased from 5 to 180 time units. This is illustrated in Figures **7.7**, **7.8**, **7.9**, and **7.10**.

The results also show that the EVRT scheme either performed the best or close to it in terms of fewest number of backups and least amount of processing time. This is because the EVRT is not affected by virtual time delays. The EVRT scheme is immune to this because its scheduling is based on an estimate of the next VRT. Recall that this estimate is based on a running average (AT as defined in Section **4.9.2**) of virtual time differences of successive messages. On the average, AT should equal MIT. The estimate of the next message's VRT is EVRT (= LVT + AT). When using the EVRT strategy a processor chooses the task with the lowest difference (LVRT − EVRT). By increasing the MIT by some quantity x we will increase both the LVRT and the AT by the same quantity x on the average. Since the EVRT scheme only depends on the difference (LVRT − EVRT) the execution sequence of the simulation will remain the same. Thus the EVRT scheme was not affected by a change of MIT. In fact the performance of the EVRT scheme remained constant as the MIT increased while the other scheme's performance became worse.

7.5.3.2 Vertical Partitioning

These experimental results showed the synchronization effect on the vertical partitioning method. Summarizing the results:

1. EVRT dramatically performed the worst
2. LVRT performed the best
3. Static partitioning was slightly worse than other schemes
4. Vertical Partitioning outperformed horizontal partitioning in the number of backups (illustrated in Figures **7.12**, **7.7**, and **7.9**) and in processing times (illustrated in Figures **7.13**, **7.8**, and **7.10**)

These results lead one to believe that vertical partitioning takes best advantage of the simulation structure. When using vertical partitioning, nodes can only send messages to nodes in the next partition. Since tasks are sorted into LVRT order (except when EVRT is used), messages are synchronized in each partition. As messages pass from partition to partition, they can only be processed if all messages within that column of nodes (partition) have a higher VRT. Processing the messages in this order reduces the number of backups.

The EVRT scheme does not produce the synchronization effect, and hence does not perform as well as the other schemes.

Static partitioning performed slightly worse than the other schemes. We attribute this to static partitioning's initial assignment of processors in all partitions. In this policy, a processor in an idle partition will immediately grab the task in its partition as soon as a message arrives. This can cause backups because messages may be processed too soon. However, dynamic repartitioning strategies assign all processors to the partition containing the input nodes. When there is no more work in a partition, a processor will move to the next partition. Therefore a processor will prefer to finish processing in its current partition before proceeding to the next partition, causing fewer backups.

We can also attribute the good results of vertical partitioning to the fact that the aggressive backup effect is not present. Since all the scheduling schemes had much higher processing times and number of backups with the horizontal partitioning method than with vertical partitioning, one is led to believe that the horizontal partitioning method was not optimal.

7.5.3.3 Application to Other Simulations

In any simulation without cycles there are precedence classes among the tasks. A precedence class is a set of tasks that are a distance of N links away from the input nodes. If a task is N links away from an input node via one path and M links via another, by convention we place the task in the precedence class that is Min (M, N) links away from an input node. Precedence class p_1 is said to be of *higher* precedence than p_2 if p_1 is fewer links away from the input nodes, while p_2 is said to be of *lower* precedence than p_1. In our 16-input-node butterfly network the group of input-nodes form the highest precedence class, p_1. All nodes that are one link from the input-nodes form the next precedence class, p_2. Continuing in this fashion all the output nodes are in the lowest precedence class, p_6. Note that these precedence classes correspond exactly to the partitions in vertical partitioning. All the tasks in partition p_i have higher precedence than all the tasks in p_{i+1}.

There is said to be a *precedence relationship* between two partitions if all tasks in one partition are of higher precedence than all tasks in the other. In vertical partitioning there is a precedence relationship between all partitions. This is not true in horizontal partitioning, since each partition contains a task from each precedence class.

Let us start with an observation about the butterfly network simulation. The way to produce the minimum number of backups is for all the processors to start working on the tasks in the highest precedence class p_1. Once this first precedence class of tasks has no more work, then the processors proceed to the next precedence class of tasks p_2 and process all of their messages until all work has been exhausted. If we continue in this fashion then there will be no backups throughout the simulation. This is true because none of the tasks within a precedence class communicate with each other so they cannot cause each other to back up (since we are assuming that the input message streams entering p_1 are monotonically increasing in virtual time), and as long as we process all the messages of one precedence class, say p_i, before going to the next precedence class p_{i+1}, then all of the messages for tasks in p_{i+1} will be sorted on the tasks' respective message queues. If all the messages ever destined for the tasks in p_{i+1} are already on the tasks' message queues before we start processing p_{i+1}, then no backup will occur in p_{i+1} (simply because if a task's input message stream is monotonically increasing in virtual time then no backups will occur).

We now generalize these effects (synchronization, aggressive backup and nodal delays) to better understand their applicability to simulations other than the butterfly network.

The source of all these effects is the partitioning. Partitioning reduces the wait time for individual processors to locate the next task to process by distributing the task queues over partitions, but at the cost of reducing the synchronization. When we introduce partitioning, each partition acts like its own small simulation. All the tasks within a partition are synchronized by the synchronization effect if we use LVRT to sort the task queues. This is true for any type of partitioning method. The problem arises when one tries to synchronize these partitions. Synchronizing partitions is done solely by the messages that pass between partitions. However, synchronizing these partitions is not an easy thing to do if there is no precedence relationship between the partitions.

In the vertical partitioning case each message proceeds through each partition in a pipeline fashion. In this case it is very clear what the precedence relationship is between partitions. Messages must first go through the input-node tasks of the input partition and traverse all the partitions in between until the messages emerge from the output partition.

The problem with horizontal partitioning is that there is no precedence relationship between the partitions. Although tasks within the horizontal partitions are synchronized, when different partitions try to synchronize with each other, additional backups occur. Another problem is that the synchronization effect within a horizontal partition goes against the precedence relationship of the tasks within the partition. In other words the tasks within a horizontal partition are forced to have their LVRTs coerced to be about the same; on the other hand, there is a specific order in which messages proceed through the tasks in the partition which is violated by this improperly induced synchronization. Furthermore, additional backups are caused by the aggressive backup effect.

Therefore the synchronization effect can be good if a simulation is partitioned such that the synchronization effect does not contradict the precedence relationship existent in the simulation. This generalization can be extended to any type of simulation that does not have feedback.

What do virtual time delays do? The important parameter is the ratio MIT:ND. (This is true for both stochastic and deterministic sources of messages because if the virtual time between successive messages is some deterministic quantity, then that deterministic quantity can be used for MIT.) Let us examine an extreme case where

$$ND > VRT_{max} - VRT_{min}$$

The quantity VRT_{max} is the maximum VRT of any message sent to any of the input nodes, and VRT_{min} is the minimum VRT of any message sent to any of the input nodes. Let us make two assumptions

1. The number of processors does not exceed the number of input nodes.
2. There is no explicit partitioning.

If this is true, no backups will occur. All of the processors will start in class p_1 and finish all the work before proceeding to the next group of tasks. Eventually all the processors will process the tasks in class p_6 and no backups will occur. The MIT probabilistically determines the values of VRT_{max} and VRT_{min}. Therefore, both the MIT and ND play an important role in this behavior.

When the ND is large in comparison to the MIT, then there will be fewer backups. This is because tasks in the higher precedence classes are more attractive, since their LVRTs are less affected by increases in the ND. Therefore, in actuality the larger the ND in comparison to the MIT the more the simulation appears to be partitioned vertically even though it may not be. As the ND increases toward the value $VRT_{max} - VRT_{min}$, fewer and fewer backups occur.

In conclusion, if there exists a precedence relationship between tasks in a simulation, then one can force the simulation system to follow this precedence relationship better by increasing the ND. For example even with the horizontal partitioning method we can improve performance by raising the ND or lowering MIT enough so that the system has implicit vertical partitions. This will not completely solve the problem in the static partitioning scheme because the partition boundaries cannot be crossed by processors.

So how do we generalize the aggressive backup effect? The aggressive backup effect occurs at partition A whenever:

1. Partition A contains tasks belonging to more than one precedence class, and
2. A task T that is not in the highest precedence class of partition A receives input messages from multiple sources, at least one of which is not within partition A, and
3. T receives a preempting message from a input message source that is not within partition A.

If these occur, all tasks in partition A that have lower precedence class than T will likely back up due to the preempting message at T. This backup can trigger additional backups. Define:

C: T and all tasks in A with lower precedence than T,

U: The preempting task, and

B: U's partition.

Following the initial backup, the tasks in C are given high priority in A due to their lower LVRT. Suppose none of the tasks in B have backed up recently; then all tasks in B have the same priority. Therefore, the tasks of C will receive a larger share of the processing power in A than U will receive in B; thus, C will eventually get ahead of U in virtual time. When this occurs, a message from U to T will cause another backup. This will not occur if all of the sources of messages to T are within A, since all the tasks within A are synchronized due to the synchronization effect (although normal backup may still occur).

We conclude that one should be careful when partitioning a simulation. If tasks in multiple precedence classes are grouped together in a partition, then tasks other than those in the partition's highest precedence class should not have any input messages from other partitions.

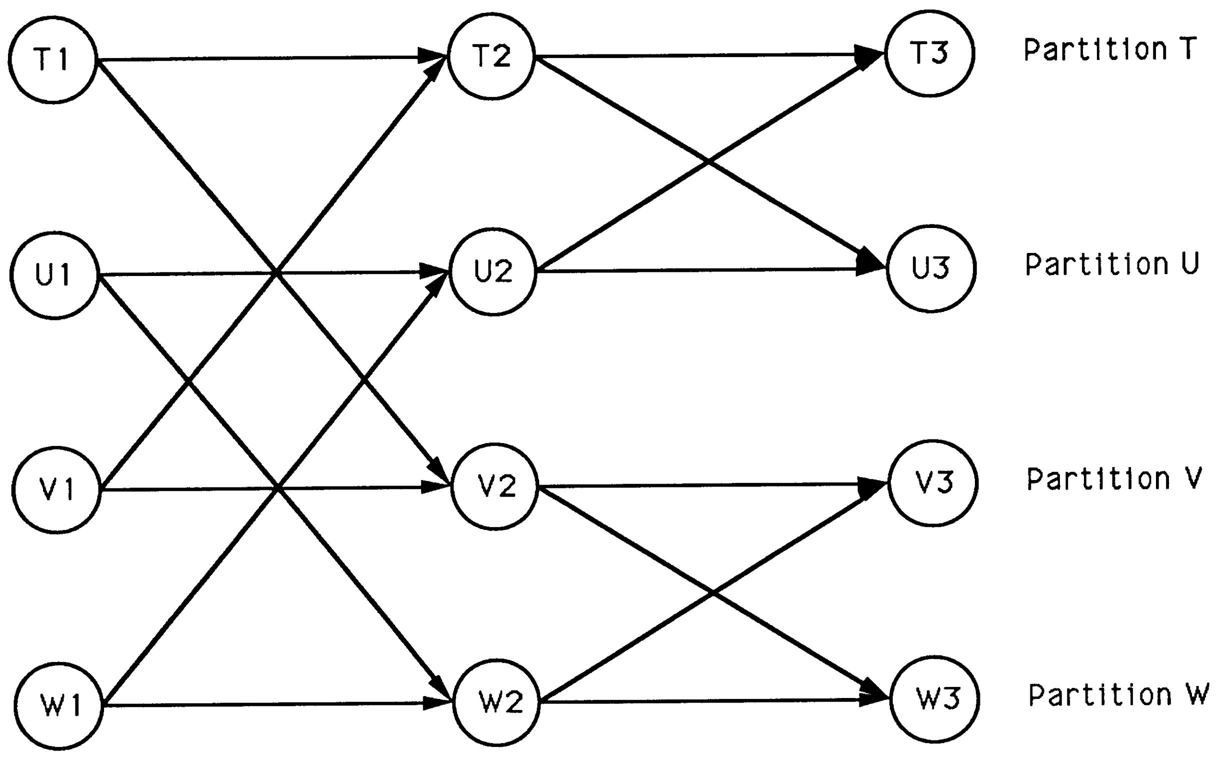
7.6. Lazy versus Aggressive Message Cancellation

Let us return to our model with its assumptions of section **7.2**. However, we will now expand our model to a 4-input-node butterfly network. We use the following modified assumptions of our model:

1. Input messages arrive at the four input nodes in increasing VRT order.
2. The destination of each input-node message is equally likely to be any of the four output nodes.
3. The VRT of each message is random based on an exponential probability distribution (pdf) based on a mean interarrival time (MIT). Each trial of the exponential pdf is rounded to the nearest whole number. The VRT of the $n + 1^{st}$ message is the VRT of the n^{th} message plus the value returned by a trial of the exponential pdf.
4. Each partition contains exactly one processor.
5. Within each partition the static partitioning scheme is used.
6. Each task has no state. Therefore, there are no conflict delays (CD) since conflict delays are implemented using state. In other words, we could make the CD=0.
7. A message's data contains the destination node address.

In this expanded model, there are four partitions T, U, V, and W as shown in Figure **7.15**. Each partition contains three tasks each, one input node, one intermediate node and one output node.

In aggressive message cancellation, when a backup occurs at a task, a set M of messages from the backward message list are removed. The messages on the backward message list were sent by this task, see Section **3.2.4**. Let us call the messages in M the preempted messages. Then a set of anti-messages AA are sent to cancel the messages in M, which had previously been sent. There is one anti-message in the set AA for each message in M. In lazy message cancellation, a message m in M is canceled only when the sending task's LVT exceeds the virtual send time of m. Let LA be the set of anti-messages that are sent for the original set of messages M in the lazy message cancellation mechanism. Therefore the set of unnecessary anti-messages is $UA = AA - LA$. The influx of the messages in UA could cause additional backups or increase processing times. This is bad.



4-Input Node Butterfly Network

Figure 7.15 4-Input-Node Butterfly Network with four Horizontal Partitions

The system benefits from lazy message cancellation if the set UA is a large subset of the set AA. An unnecessary anti-message occurs at a task I when:

1. A preempted message sent by I is canceled and then sent again unchanged, which will always happen in our model since tasks have no state.
2. A preempted message sent by I and the messages sent by I in response to its preempting message go to different destinations.

We expect lazy message cancellation to greatly outperform aggressive message cancellation whenever a task is repeatedly backed up. Such a situation occurs when the aggressive backup effect occurs.

The following example is contrived to illustrate the two sources of unnecessary anti-messages on the network in Figure 7.15. The example is a snapshot of a simulation in

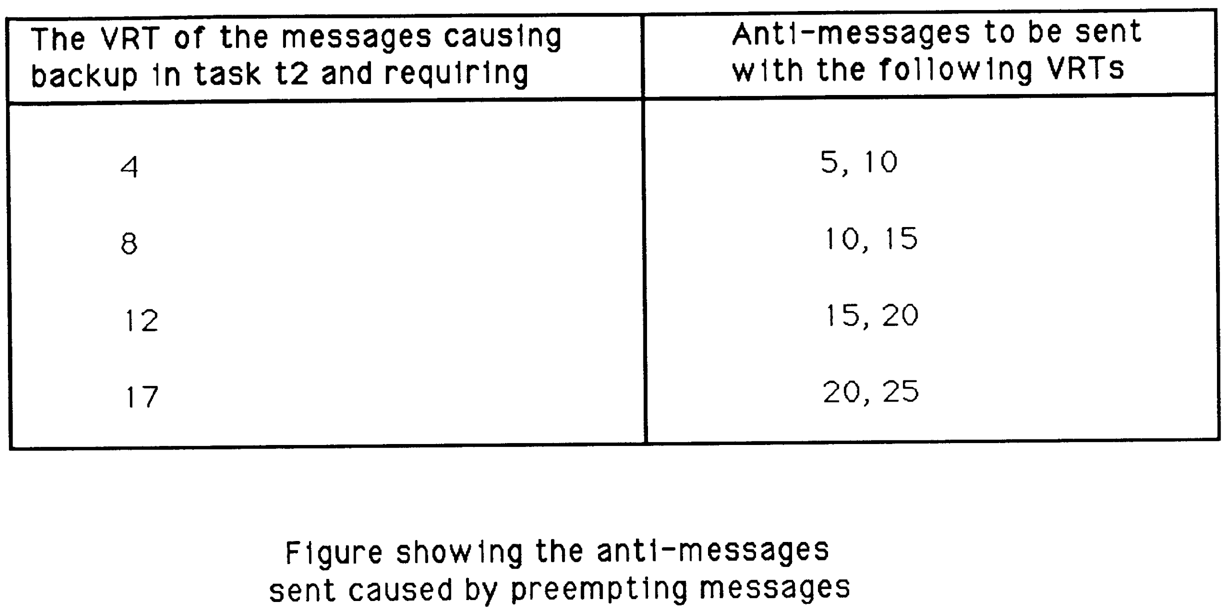
VRTs of Processed Messages at T2	VRTs of Messages on the Input Message Queue at T2
5	(10, 15, 20, 25)
5, 10	(15, 20, 25)
4*	(5, 10, 15, 20, 25)
4, 5	(10, 15, 20, 25)
4, 5, 10	(15, 20, 25)
4, 5, 10, 15	(20, 25)
4, 5, 8*	(10, 15, 20, 25)
4, 5, 8, 10	(15, 20, 25)
4, 5, 8, 10, 15	(20, 25)
4, 5, 8, 10, 15, 20	(25)
4, 5, 8, 10, 12*	(15, 20, 25)
4, 5, 8, 10, 12, 15	(20, 25)
4, 5, 8, 10, 12, 15, 20	(25)
4, 5, 8, 10, 12, 15, 20, 25	()
4, 5, 8, 10, 12, 15, 17*	(20, 25)
4, 5, 8, 10, 12, 15, 17, 20	(25)
4, 5, 8, 10, 12, 15, 17, 20, 25	()

* - Processing the message with this VRT caused a backup at T2

Figure 7.16 The Aggressive Backup Effect Showing Messages Processed and Message Queues for each Task

progress. The details of how the snapshot arises are omitted. Suppose that partition T gets ahead of partition V in virtual time. In other words, the LVTs of the tasks in partition T are larger than those of partition V. Suppose the following messages are on

1. t_2's input queue: ((5, t_3) (10, t_3) (15, t_3) (20, t_3)) (25, t_3)), and
2. v_1's input queue: ((3, v_3) (4, u_3) (6, v_3) (7, v_3) (8, u_3) (9, v_3) (11, v_3) (12, u_3)



The VRT of the messages causing backup in task t2 and requiring	Anti-messages to be sent with the following VRTs
4	5, 10
8	10, 15
12	15, 20
17	20, 25

Figure showing the anti-messages sent caused by preempting messages

Figure 7.17 The VRT of Preempting Messages and the VRTs of their Anti-Messages

(13, v_3) (14, v_3) (17, u_3)).

Figure **7.16** shows a possible execution sequence of messages at task t_2. Each entry shows the messages processed and the input message queue for t_2 immediately after t_2 has been processed by T's processor. Note that there is time between each entry where T's processor may be processing messages at t_3 that were just sent by t_2. We assume there are no messages in $t_1's$ message queue so no processing time is spent on t_1. The messages at t_2 are processed faster than at v_1 because V's processor spends some of the example processing messages with VRT=3, 6, 7, 9, 11, 13 and 14 at tasks v_2 and v_3 whereas T's processor only spends additional time processing messages at t_3. Here the task t_2 backed up at virtual times 4, 8, 12, and 17. In this diagram we see that there are four preempting messages to t_2, at times 4, 8, 12, and 17. In this case all messages that cause t_2 to back up are sent onward to u_3.

In the aggressive message cancellation case, after each backup, t_2 sends anti-messages for each of the preempted messages shown in Figure **7.17**. Now when t_2 repeatedly backs up due to the aggressive backup effect it will also repeatedly send out additional sets of anti-messages, many of which may be unnecessary. In our example

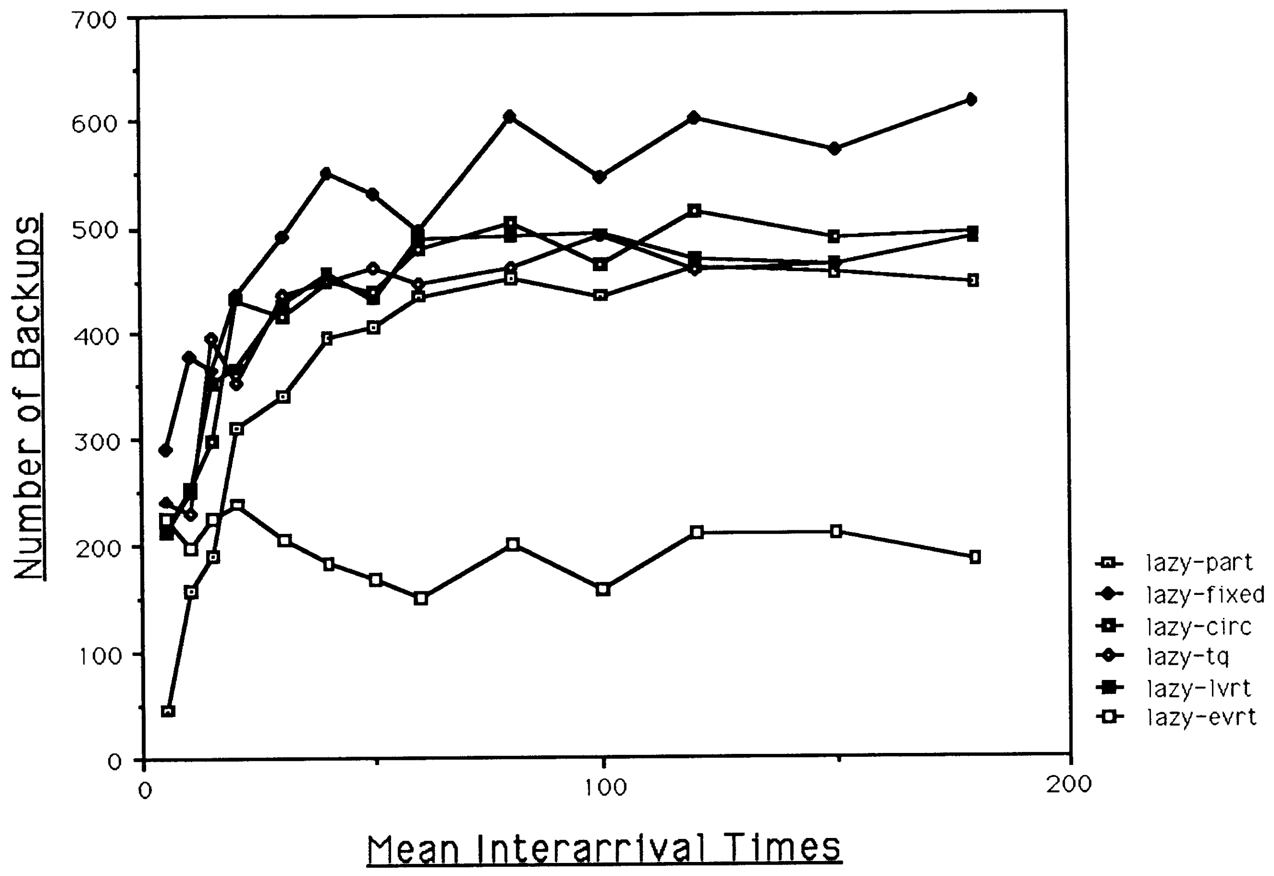


Figure 7.18 Number of Backups versus MIT for Lazy Cancellation for H_8

the preempted messages with VRTs of 10, 15 and 20 were canceled unnecessarily, since the same messages are later resent. The preempted message with VRT of 10, 15 and 20 are processed and canceled twice. These additional anti-messages could cause more backups than those caused by the original preempting messages at times 4, 8, 12, and 17. This is an example of the first kind of unnecessary anti-message and is called the repeated backup effect.

However in the lazy message cancellation case none of these additional anti-messages are sent. Therefore, only the backups caused directly by the preempting messages 4, 8, 12, and 17 would cause any additional backups at t_3 or u_3. This is very important since after passing through the backed-up task, the set of preempting messages, traveling from v_1, go to u_3, while the set of preempted messages, traveling

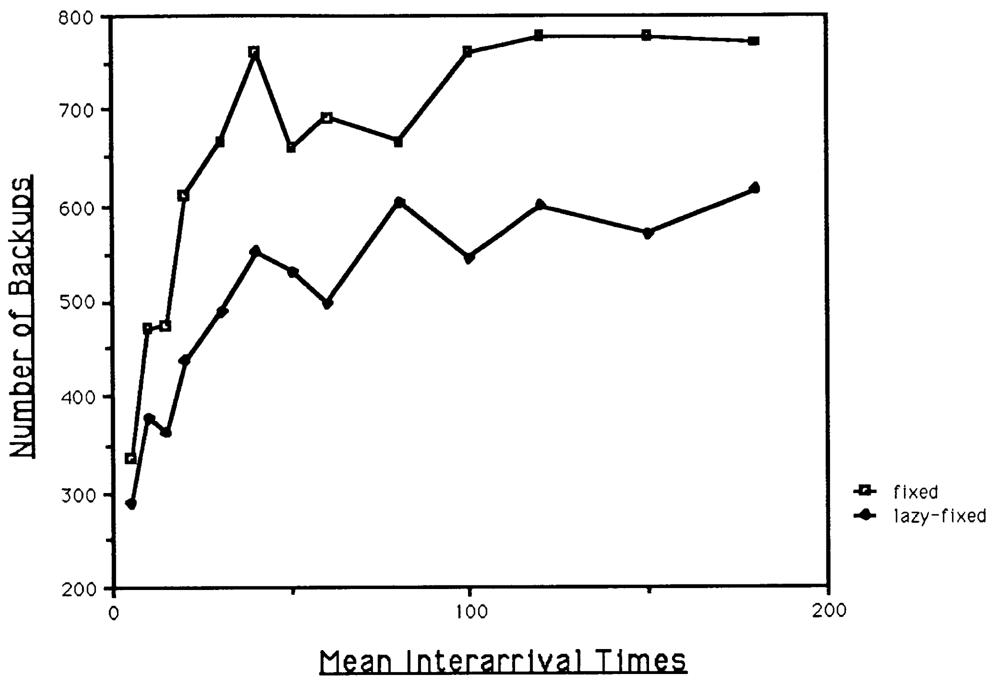


Figure 7.19 Number of Backups versus MIT for H_8 using Aggressive or Lazy Cancellation with Fixed-list scheduling

from t_1, go to t_3. Since the preempted messages and the preempting messages travel to different destinations, the anti-messages for the preempted messages never need to be sent. The sending of these anti-messages in the aggressive case is an example of the second kind of unnecessary anti-messages.

Let us explain what is happening with aggressive message cancellation. If we assume:

1. A preempting message m_1 from v_1 forces t_2 to back up. This causes t_2's LVT to be synchronized with the tasks in partition V.
2. m_1 and all subsequent messages from v_1 to t_2 are destined for u_3.

Note that even though m_1 is destined for u_3, t_3 is likely to back up if t_2 sends antimessages to t_3 due to t_2's backup.

One thing that occurs is the aggressive backup effect. This happens when T's processor gives priority to the backed-up task (t_2) or tasks (t_2 and t_3). This will allow

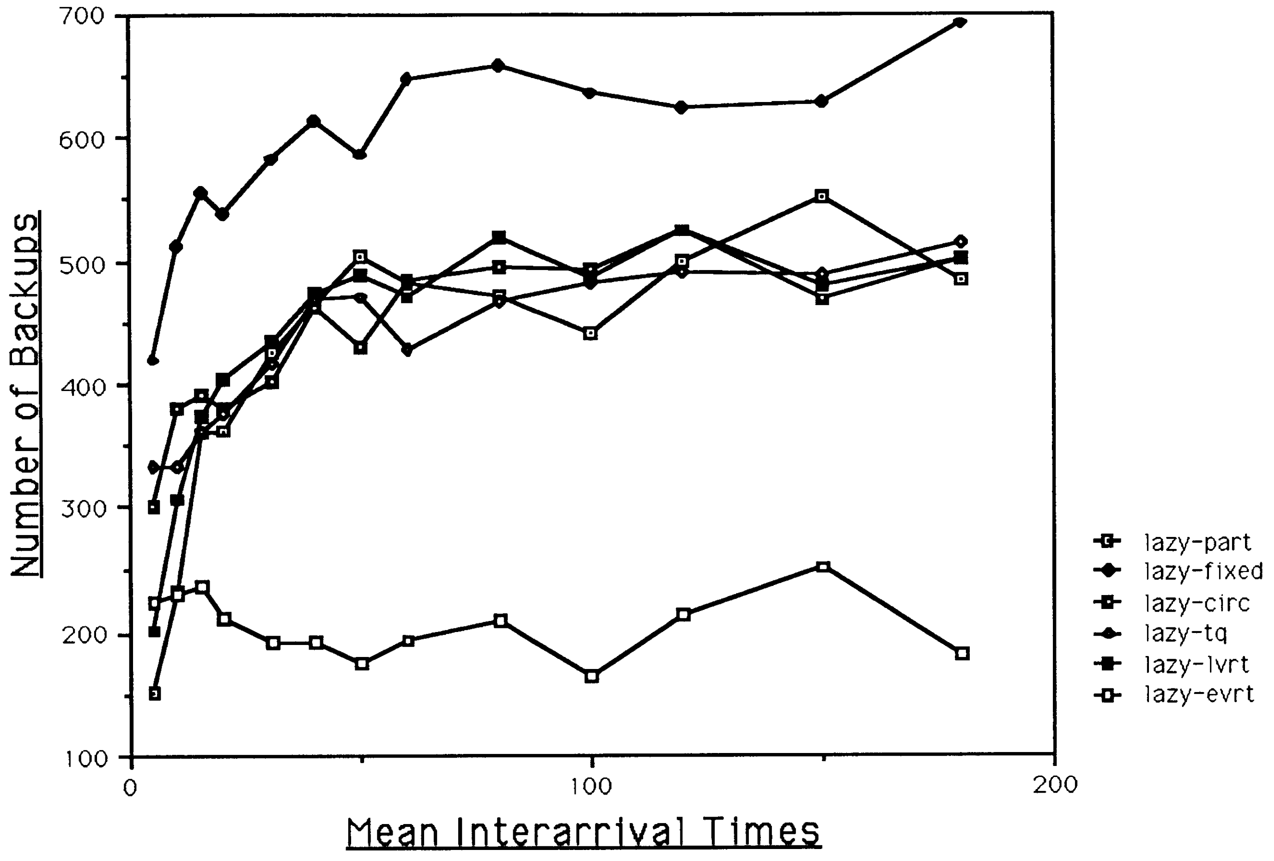


Figure 7.20 Number of Backups versus MIT for Lazy Cancellation for H_{16}

the backed-up task or tasks to get ahead (in virtual time) of the tasks in partition V. Once this occurs any new message from v_1 to t_2 will again cause backup at t_2.

Now the repeated backup effect occurs. Once again, t_2 may send anti-messages to t_3 causing backup even though none of the preempting messages are destined for t_3. Furthermore, the same preempted messages are likely to be canceled and later resent to t_3 when repeated backups occur at t_2. The preempted messages remain the same because preempting messages do not change tasks' states and because there are no conflict delays. Therefore, the additional anti-messages to t_3, the resent preempted messages to t_3 and the resulting backups at t_3 are all unnecessary.

In lazy message cancellation, these preempted messages are never canceled because messages are never canceled until the task is sure that the message will not be re-sent.

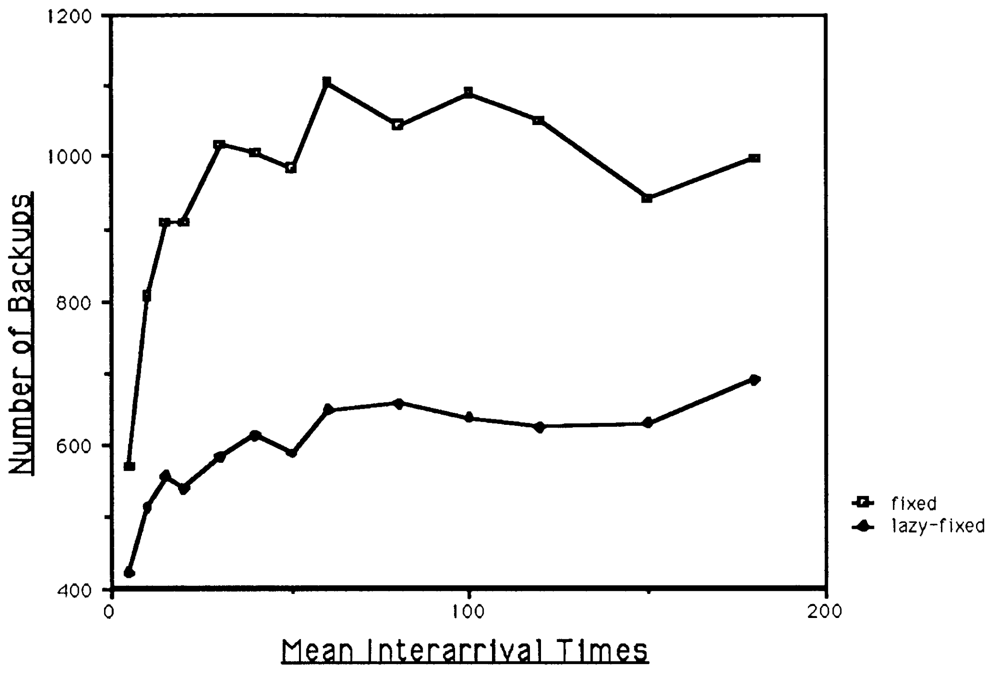


Figure 7.21 Number of Backups versus MIT for H_{16} using Aggressive or Lazy Cancellation with Fixed-list scheduling

In this way, none of the preempted messages from t_2 to t_3 would be repeatedly sent and canceled by t_2. The problem that occurs here is that, in the aggressive case, t_3 was forced to repeatedly back up unnecessarily. Thus, in this example lazy message cancellation would prevent these additional backups.

7.6.1 Experiments

In these experiments we studied the differences between lazy and aggressive message cancellation. Each of these experiments was executed multiple times with different MITs. In all the experiments each of six different scheduling policies was executed at least twice. These scheduling policies were static partitioning, fixed-list, circular-list, longest task queue, lowest LVRT and EVRT dynamic repartitioning. Each scheduling policy was used with lazy as well as aggressive message cancellation.

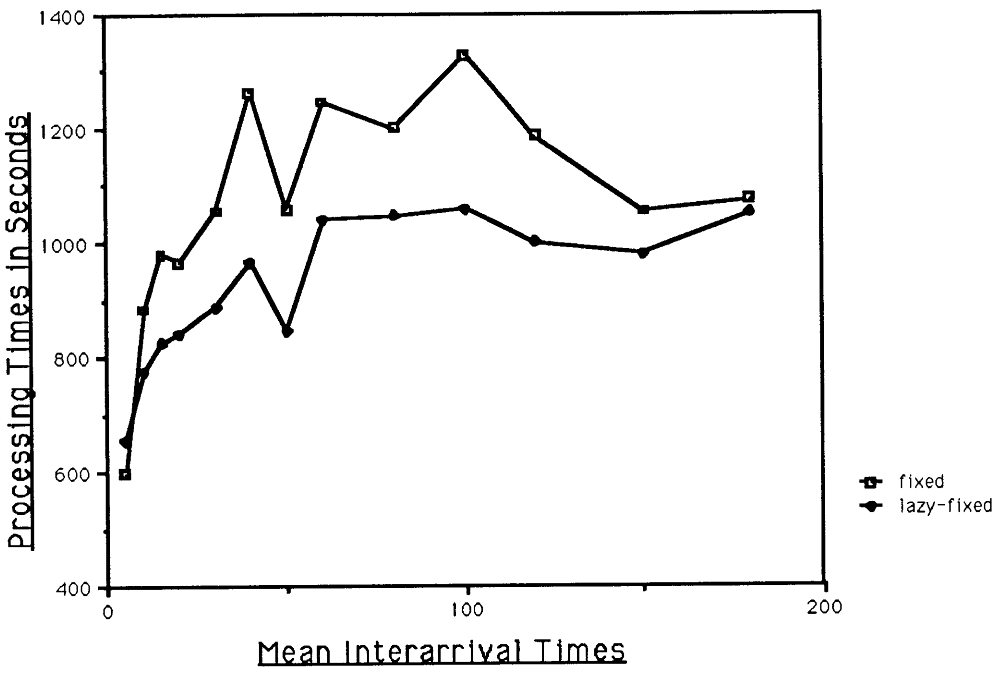


Figure 7.22 Processing Time versus MIT for H_{16} using Aggressive or Lazy Cancellation with Fixed-list scheduling

7.6.1.1 H_8-Partition Case

Figure **7.18** shows the number of backups for each of the scheduling policies while using lazy message cancellation as the MIT increases. The policies are static partitioning (lazy-part), fixed-list (lazy-fixed), circular-list (lazy-circ), longest task-queue (lazy-tq), lowest LVRT (lazy-lvrt), and lowest EVRT (lazy-evrt). The EVRT scheme had the fewest backups, whereas the fixed-list scheme had the most. Lazy message cancellation reduces the number of backups in this partitioning method for all of the scheduling schemes. Figure **7.19** illustrates this effect by showing the number of backups for the fixed-list policy using aggressive (fixed) and lazy (lazy-fixed) message cancellation. In comparing Figure **7.7** and **7.18**, it is clear that lazy message cancellation decreases the backups in all of the scheduling schemes. The figures also show that lazy message cancellation preserves the performance order of the scheduling schemes. That is, if a scheduling scheme has fewer backups with aggressive message cancellation than

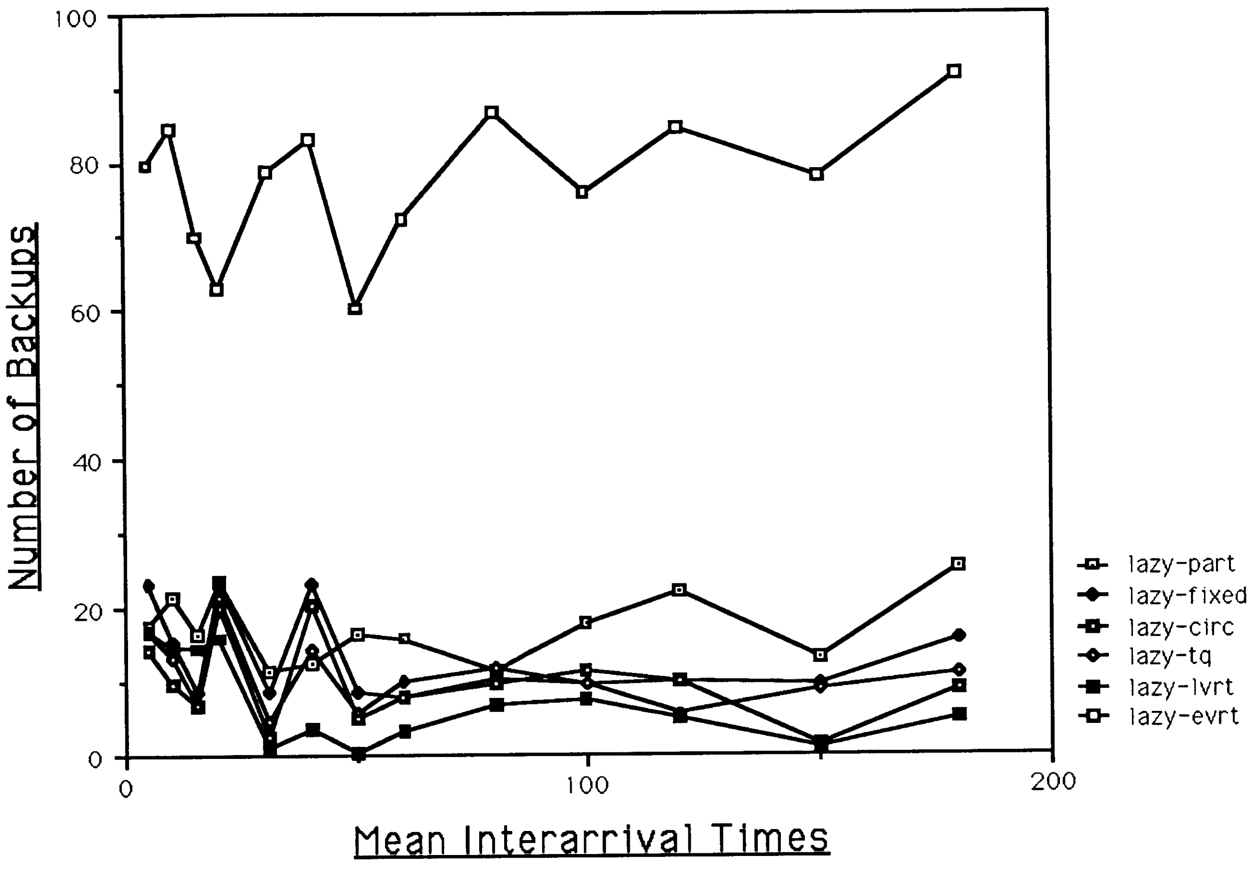


Figure 7.23 Backups versus MIT for Lazy Cancellation

another, then it also has fewer backups with lazy message cancellation.

7.6.1.2 H_{16}-Partition Case

Figure **7.20** shows the number of backups for each of the scheduling policies while using lazy message cancellation as the MIT increases. Lazy message cancellation again reduces the number of backups. The effect for the fixed-list strategy is shown in Figure **7.21**. In comparing Figure **7.9** and **7.20**, lazy message cancellation again decreases the backups in all of the scheduling schemes, while preserving the performance order. Lazy message cancellation reduced the processing time of the fixed-list strategy only; this is shown in Figure **7.22**.

7.6.1.3 V_6-Partition Case

Lazy message cancellation had essentially no effect on either processing times or backups for vertical partitioning. Figure **7.23** shows the number of backups when lazy message cancellation is used. These results are comparable to those results shown in Figure **7.12** where aggressive message cancellation is used.

7.6.2 Discussion

These results showed that for horizontal partitioning, lazy message cancellation improved the performance by either decreasing processing time or decreasing backups. In the best case, lazy message cancellation can reduce the number of backups by nearly 50%. This additional performance costs essentially nothing since the processing time for a simulation using aggressive message cancellation was about the same as for lazy message cancellation. In vertical partitioning, however, lazy message cancellation had no effect because the aggressive backup effect is not present.

Our data confirm the hypothesis that lazy message cancellation has a greater effect when a task continually backs up: specifically, when the aggressive backup effect is present. The aggressive backup effect is present in horizontal partitioning; consequently, lazy message cancellation reduced the number of backups and processing times. However, the aggressive backup effect is not present in vertical partitioning, and so lazy message cancellation had little effect.

7.6.3 Application to Other Simulations

Lazy message cancellation:

1. Does not unnecessarily cancel messages.
2. Does not repeatedly cancel messages.

These properties of lazy message cancellation are good because they help to prevent the repeated backup effect. This occurs whenever a task repeatedly backs up such that messages are repeatedly canceled, as in the aggressive backup effect. When the aggressive backup effect occurs, repeated backups in the original task allow messages to be canceled repeatedly, causing other tasks to back up repeatedly.

We conclude that lazy message cancellation is good when the partitioning method forces tasks to repeatedly back up. Otherwise lazy message cancellation has no effect on either processing times or backups. However, since lazy message cancellation does not cost much in terms of processing time, it appears that one should always use it in the network simulation.

Our original discussion of lazy message cancellation did not consider state changes at the backed-up task, as are needed, for example, to implement conflict delays. Whenever the preempting message causes a change in the data field of the output message or a state change in the backed-up task, it may cause a different sequence of output messages to be sent by the backed-up task. If this occurs, then it is possible for the preempting messages to a task to affect the preempted messages sent by the task even if they are destined for different nodes. If a state change occurs at the backed-up task, then anti-messages are unnecessary only if the preempted messages are canceled and resent without any change in contents.

Under some circumstances, lazy message cancellation can cause more messages and backups than aggressive message cancellation. This is possible if lazy message cancellation delays needed backups, which could increase the number of backups. Section **7.13.2** explains why lazy message cancellation can delay backups which in turn causes more backups.

7.7. Real-Time Delay

The performance of the simulation is dependent not only on the inter-message delays (IM) but also on the processing time delays (PT). The inter-message delay is the real-time delay between messages, while the processing time delay is the real time needed to process a message at a node. We show that the ratio $\frac{IM}{PT}$ governs the performance of the simulation.

We return to our two-input-node network model with four tasks and its accompanying basic assumptions from Section **7.2**. The model was partitioned vertically as shown in Figure **7.2** and horizontally as shown in Figure **7.1**. In this section we vary IM and PT to study their effect on performance. In our model we assume that 30 input messages arrive at each of the two input nodes in increasing VRT order. Since we assume the input streams of messages are increasing monotonically in virtual time, they cannot possibly force a backup at the input nodes. This is not true at the output nodes. We wish to study how the real-time delays affect backups at the output nodes.

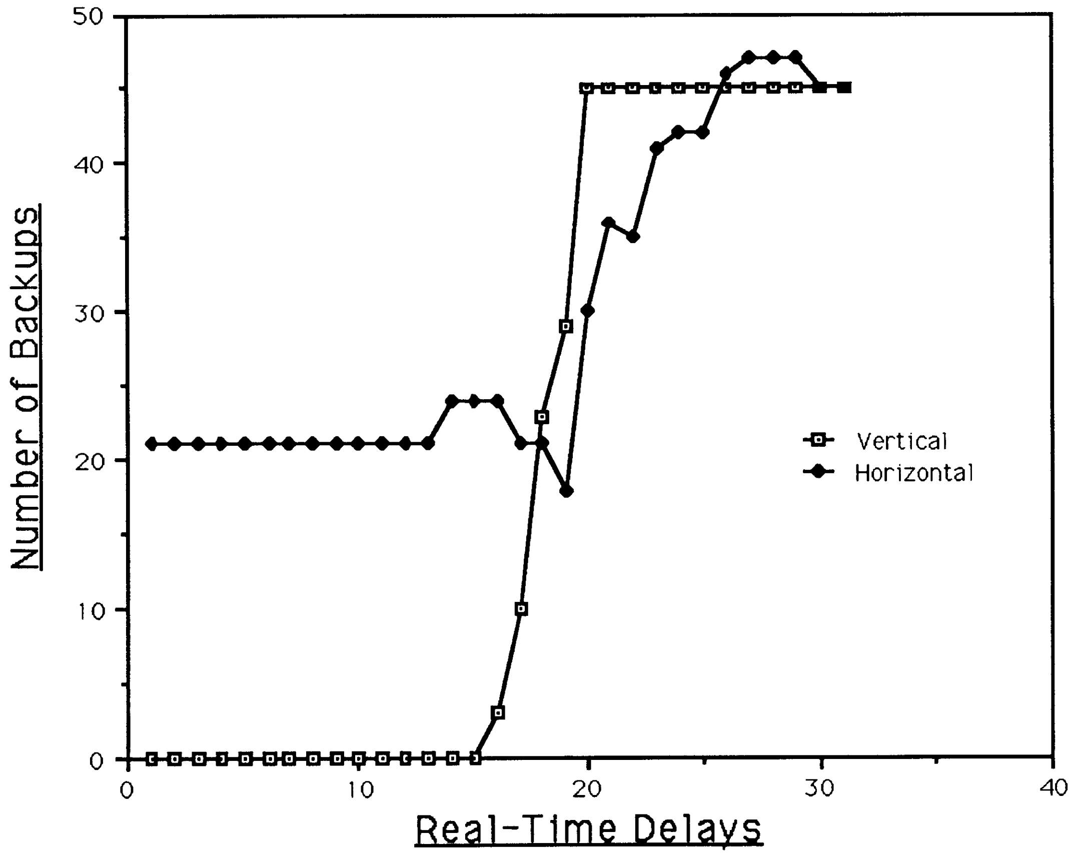


Figure 7.24 Real-Time Delay Effects with PT=10 with varying IM

In our experiment we fixed PT = 10, while we varied the values of IM between 1 and 31. The results are illustrated in Figure **7.24**. The performance of the simulation is measured in the number of backups. In this figure, *Vertical* indicates the number of backups for vertical partitioning and *Horizontal* indicates the number of backups for horizontal partitioning.

The results show that real-time delays of messages can have a dramatic effect on backup performance of our simulations. Figure **7.24** shows that horizontal partitioning always has backups whereas vertical partitioning has no backups until the $\frac{IM}{PT}$ ratio exceeds 1.5. The data show that the vertical partitioning case is more susceptible to a changing $\frac{IM}{PT}$ ratio than the horizontal partitioning. For example, by varying the IM between 15 and 20, vertical partitioning went from 0 to 45 backups while horizontal partitioning only went from 24 to 30 backups. Although both methods of partitioning have about 45 backups when the ratio of $\frac{IM}{PT}$ is extremely high, horizontal partitioning is less susceptible to small variations.

In our experiments on Concert, horizontal partitioning always had many times more backups than vertical partitioning. This suggests that the range of $\frac{IM}{PT}$ in the Concert experiments is between 0 and 1.6. To determine how varying the $\frac{IM}{PT}$ ratio affects our simulation we will study their effects on vertical and horizontal partitioning separately.

7.7.1 Vertical Partitioning

In this section we study the effects of varying the ratio of $\frac{IM}{PT}$. We show that the number of backups is highly influenced by the order of messages in the input partition's output message stream.

Recall our notion of streams from Section **7.2**. The input and output messages streams for vertical partitioning are illustrated in Figure **7.25**. Each input task has an output stream that is merged in real time into the input partition's output stream. The input partition's output stream is the same as the output partition's input stream. Each message of this input stream goes to one of two output tasks' input streams depending on its destination.

We only determine the number of backups at the output nodes of our model, since we know that the input nodes will not have any backups. Backups will not occur

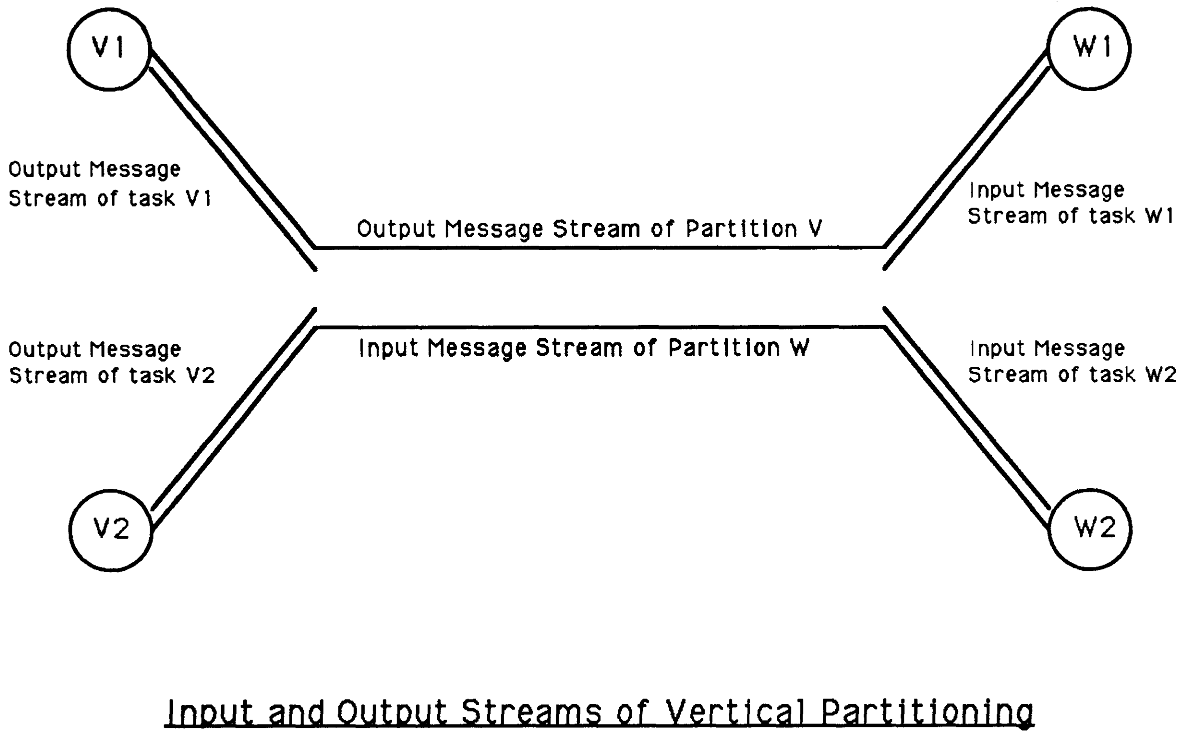

V1
Output Message Stream of task V1
Output Message Stream of Partition V
W1
Input Message Stream of task W1
Output Message Stream of task V2
Input Message Stream of Partition W
Input Message Stream of task W2
V2
W2
Input and Output Streams of Vertical Partitioning

Figure 7.25 Input and Output Message Streams for Vertical Partitioning

at the output partition when the input stream to each output node is monotonically increasing in virtual time. This is guaranteed whenever the output message stream from the input partition is monotonically increasing. This is not equivalent to the output message streams' from the input nodes being monotonically increasing. The output message stream of the input partition is monotonically increasing if and only if:

1. The output message streams of the input nodes are monotonically increasing, and
2. The input tasks are synchronized. In other words, only the task with the lowest LVRT can output a message at any given time.

The first item is always true since the input nodes never back up. The second item will hold under certain $\frac{IM}{PT}$ ratios for vertical partitioning.

If the output stream from the input partition is not monotonically increasing, backups are likely to occur. The only case where backups do not occur is when the input streams of the output nodes are still monotonically increasing. This occurs in the somewhat unlikely event that all the messages that are out of virtual time order in the input partition's output stream go to separate destinations. Therefore, an output stream that is not monotonically increasing will not necessarily cause backups, but makes backups more likely.

Figure **7.2** shows a diagram of vertical partitioning on the four-node model. When examining the real-time delays for vertical partitioning, three categories of operation exist. They are:

1. $\frac{IM}{PT} < 1$, and
2. $\frac{IM}{PT} > 2$, and
3. $2 > \frac{IM}{PT} > 1$.

7.7.1.1 $\frac{IM}{PT} < 1$

Whenever $\frac{IM}{PT} < 1$, messages arrive at the input nodes faster than the processor can process them; therefore, a backlog of messages forms on the input message queues. Let $lvrt_1$ be input node v_1's LVRT and $lvrt_2$ be input node v_2's LVRT. Since both input nodes' message queues contain messages and there is only one processor in this partition, the processor can safely processes the input node with the lowest LVRT. Without loss of generality let us say that the input node with the lowest LVRT is v_1 and its first message is m_1. After processing v_1, it will output the message to an output node after a virtual time delay ND. We claim that any future messages processed at either input node will have a higher LVRT because:

1. At the time m_1 is processed, none of the messages on v_1's message queues has a lower VRT than m_1.
2. At the time m_1 is processed, none of the messages on v_2's message queues has a lower VRT than m_1.
3. Any future message to arrive at the input nodes have a higher VRT than m_1.

Statement 1 is true because the message queues are sorted in virtual time order, and v_1 always processes the message with the lowest VRT off its message queue first. Statement 2 is true because there is only one processor in this partition and it always chooses to process the task with the lowest LVRT. Finally, since the input message streams to the input nodes are monotonically increasing, any new messages that arrive to a task v_i must have a higher VRT than any of the messages on v_i's message queue. Thus, statement 3 is true.

Since the message processed by the lone processor is always the one with the lowest VRT from that point in real time onward, the output message stream from the input partition is always monotonically increasing. Therefore, the output message stream of v as well as the input message streams of w_1 and w_2 will be monotonically increasing. This is the safe case, since processing the task with the lowest LVRT can never cause backup at the output partitions.

7.7.1.2 $\frac{IM}{PT} > 2$

In this category the processor in partition v waits for two messages to simultaneously arrive at the empty messages queues of both the input nodes at the start of the simulation. This is true in our model, but this synchronization is unlikely to occur in real simulations because the IM between successive messages will never be exactly the same. In real simulations, these fluctuations in IM will only affect the results of this analysis when $\frac{IM}{PT} = 1$ or 2. In these cases, the loss of synchronization may cause a few additional backups.

When the messages arrive, the processor:

1. Processes the message with the lowest VRT at one of the input nodes, say v_i, and sends the message on toward its destination. When this is complete, v_i's input message queue will be empty.
2. Processes the message with the next lowest VRT at v_j. When this is complete, both input message queues will be empty.
3. Waits for the next set of messages, one for each input node.
4. Returns to step 1 (when the new messages arrive).

Thus the processor in the input partition will process messages off each of the input message streams half the time. This is the unsafe case, since processing v_j's message in step 3 may cause backup if the next message to v_i has a lower VRT and they both have the same destination output node, say w_i. In this case the input stream to w_i will not be monotonically increasing and backup will occur at w_i.

On the other hand if the destinations of the two messages are different and the input streams to w_1 and w_2 have been monotonically increasing, the streams will still be monotonically increasing after the messages are sent. Eventually, it is likely that after some iteration of step 3, the processing of v_j's message will cause a backup because the next message to v_i will have a lower VRT and the same destination.

7.7.1.3 $2 > \frac{IM}{PT} > 1$

This case consists of behavior from both of the previous categories. Whenever a processor finishes processing a task, the input queues of the input tasks will either be:

1. Both non-empty, or
2. One or both empty.

The first case corresponds to the safe case as in category 1 and no backups will be caused by processing the message with the lowest LVRT. The second case corresponds to the unsafe case as in category 2 and backups will occur under the conditions stated in category 2. When we start a simulation, the unsafe case will occur because the task processing the first message will have an empty queue after the first message is processed. As the simulation proceeds, both input queues will eventually become non-empty because messages arrive in the system faster than they are serviced (2 messages arrive every IM, 1 message is serviced every PT). Therefore the safe case will become more common.

Once the simulation has entered the safe case, it is possible that one of the input queues will become empty. Suppose that most of the messages queued at one task v_1 have lower LVTs than the messages queued at the partition's other task v_2. Here, the processor will mostly work at v_1. Therefore, messages arrive at v_1 slower than they are processed at v_1. Therefore, v_1's queue will eventually be empty, and hence the unsafe case will arise again. Eventually the safe case must arise again and this safe-unsafe alternation continues. As the simulation proceeds it seems likely that the input queues will slowly build up, so that the unsafe case becomes progressively rarer.

7.7.2 Horizontal Partitioning

Here we study the effects of varying the ratio $\frac{IM}{PT}$ with horizontal partitioning. Again, we study the number of backups at only the output nodes, because the input nodes do not back up. Backups will only occur when the input stream to an output node is not monotonically increasing.

Note that in the horizontal case, the two input nodes are in different partitions; therefore, each has a different processor. Figure **7.1** shows a diagram of horizontal partitioning on the four-node model. When examining the real-time delays for horizontal partitioning, three categories of operation exist. They are:

1. $\frac{IM}{PT} < 1$,
2. $2 > \frac{IM}{PT} > 1$
3. $\frac{IM}{PT} > 2$

7.7.2.1 $\frac{IM}{PT} < 1$

Whenever $\frac{IM}{PT} < 1$ messages arrive at the input nodes faster than the processors can process them; therefore, a backlog of messages forms on the input message queues. This prevents any backups due to real-time delays, but backups are still caused by inter-partition messages (the aggressive backup effect), and are aggravated by virtual-time delays. These backups are always present no matter what the ratio $\frac{IM}{PT}$ happens to be. Figure **7.26** illustrates an example where backup occurs and $\frac{IM}{PT} < 1$. In all of the examples in this section we use virtual-time delays that are large with respect to the nodal delay (ND) by setting ND=0. In this example backup occurs because task t_2 gets ahead in virtual time, so that backup results when task u_1 sends a message to t_2 with a low VRT.

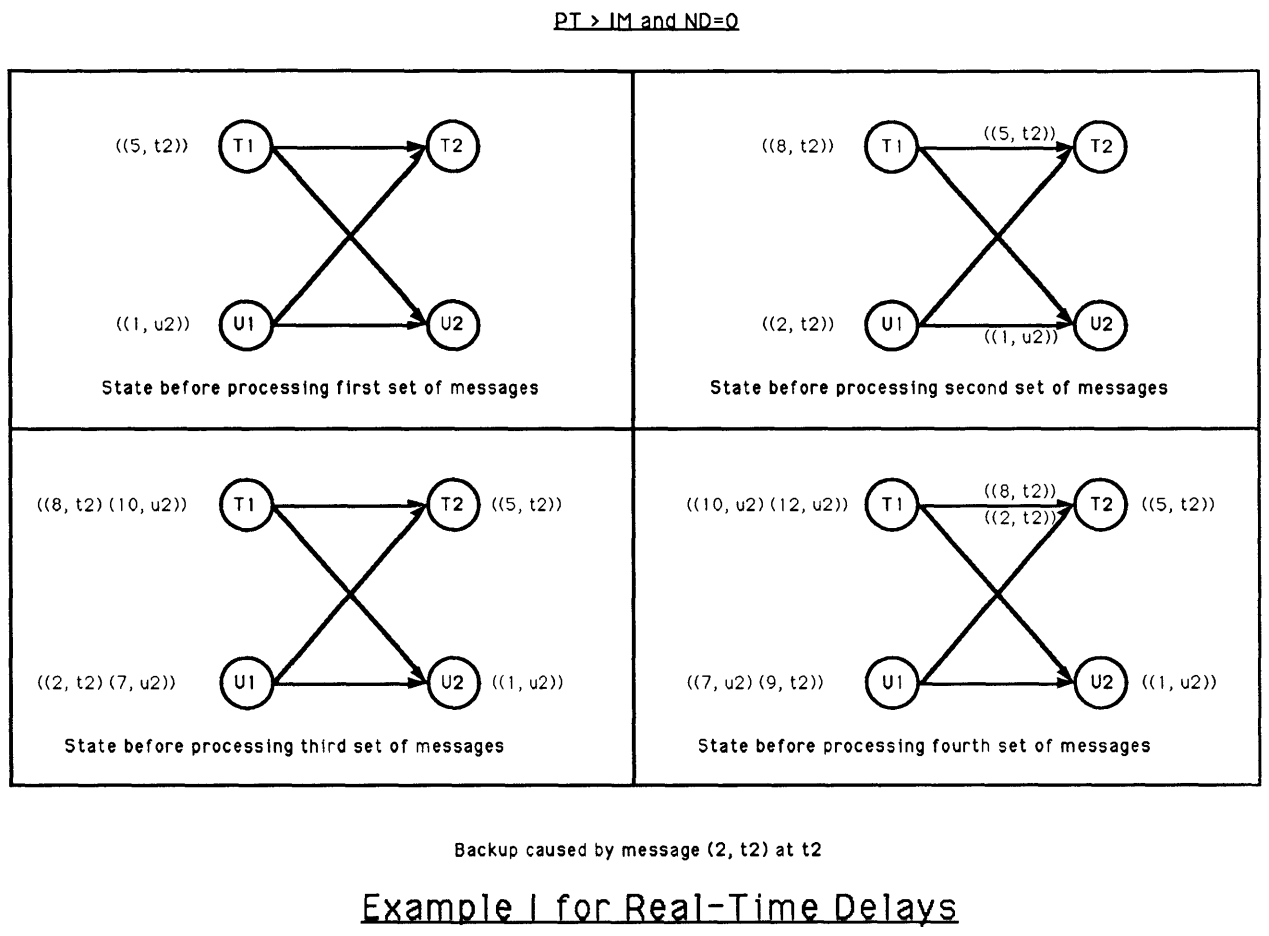
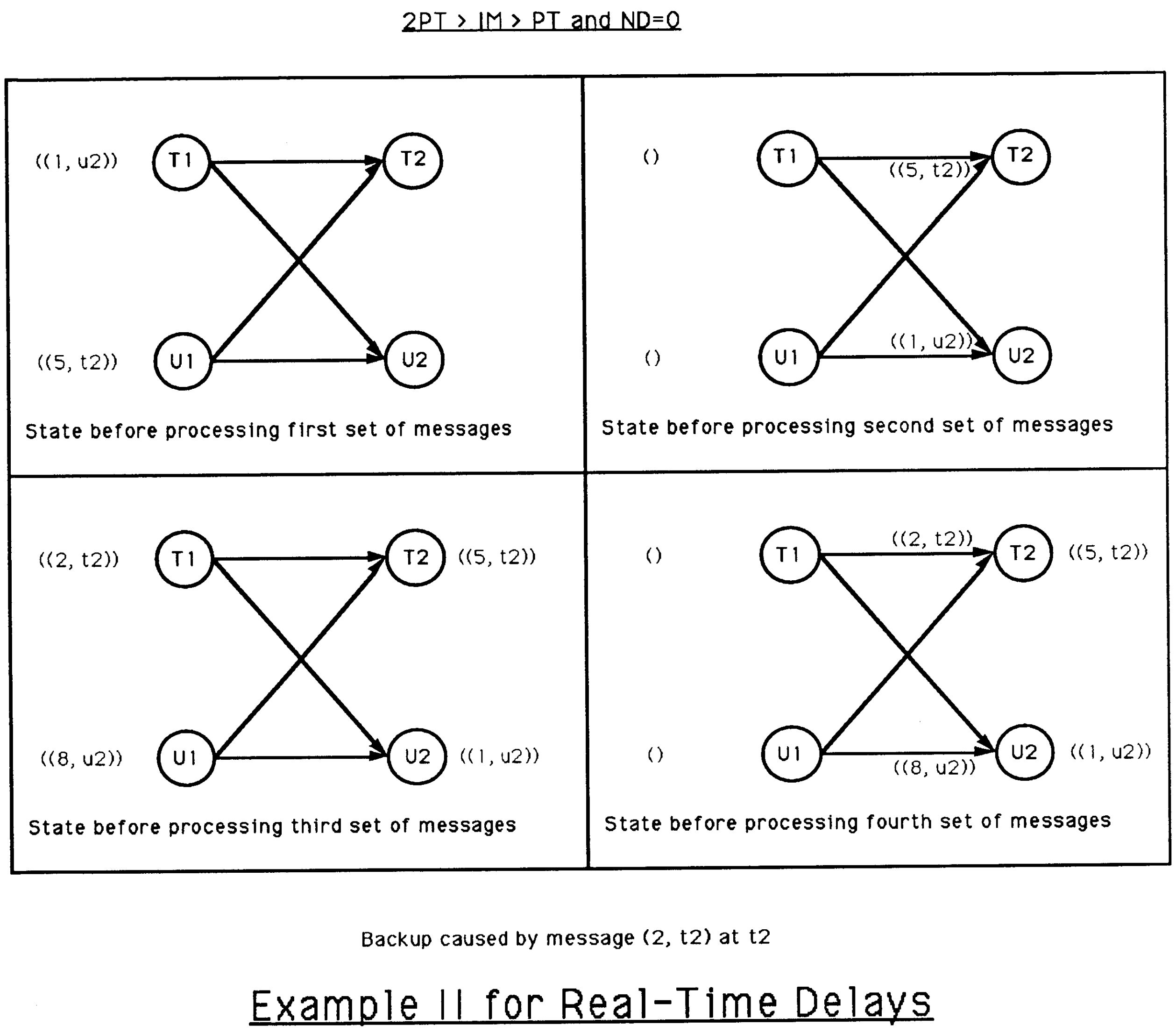

PT > IM and ND=0
((5, t2))
((1, u2))
T1
T2
U1
U2
State before processing first set of messages
((8, t2))
((2, t2))
((5, t2))
((1, u2))
State before processing second set of messages
((8, t2) (10, u2))
((2, t2) (7, u2))
((5, t2))
((1, u2))
State before processing third set of messages
((10, u2) (12, u2))
((7, u2) (9, t2))
((8, t2))
((2, t2))
((5, t2))
((1, u2))
State before processing fourth set of messages
Backup caused by message (2, t2) at t2
Example I for Real-Time Delays

Figure 7.26 Horizontal Partitioning Example I

7.7.2.2 $2 > \frac{IM}{PT} > 1$

Whenever $2 > \frac{IM}{PT} > 1$, then it is possible that either t_1 or u_1's message queues could be empty after each processor processes one task because processors process messages faster than they arrive. However, after a processor processes two tasks, the input node's message queue for the processor's partition must be non-empty. Backup is even more likely than in the first case. Figure **7.27** illustrates an example where backups occur and $2 > \frac{IM}{PT} > 1$. The problem in this example is that task t_2 gets ahead in virtual time. This is because the processor for partition t must process task t_2's message with VRT=5 since t_1 has an empty message queue. Now when the next



2PT > IM > PT and ND=0
((1, u2))
T1
T2
((5, t2))
U1
U2
State before processing first set of messages
()
((5, t2))
((1, u2))
()
State before processing second set of messages
((2, t2))
((5, t2))
((8, u2))
((1, u2))
State before processing third set of messages
()
((2, t2))
((5, t2))
()
((8, u2))
((1, u2))
State before processing fourth set of messages
Backup caused by message (2, t2) at t2
Example II for Real-Time Delays

Figure 7.27 Horizontal Partitioning Example II

message arrives at t_1 with a lower VRT than the message just processed at t_2, backup will occur. Since one partition is generally ahead of the other in virtual time, this phenomenon can always occur whenever ND is small. Note that this example will not back up when $\frac{IM}{PT} < 1$, but example I will back up when $2 > \frac{IM}{PT} > 1$!

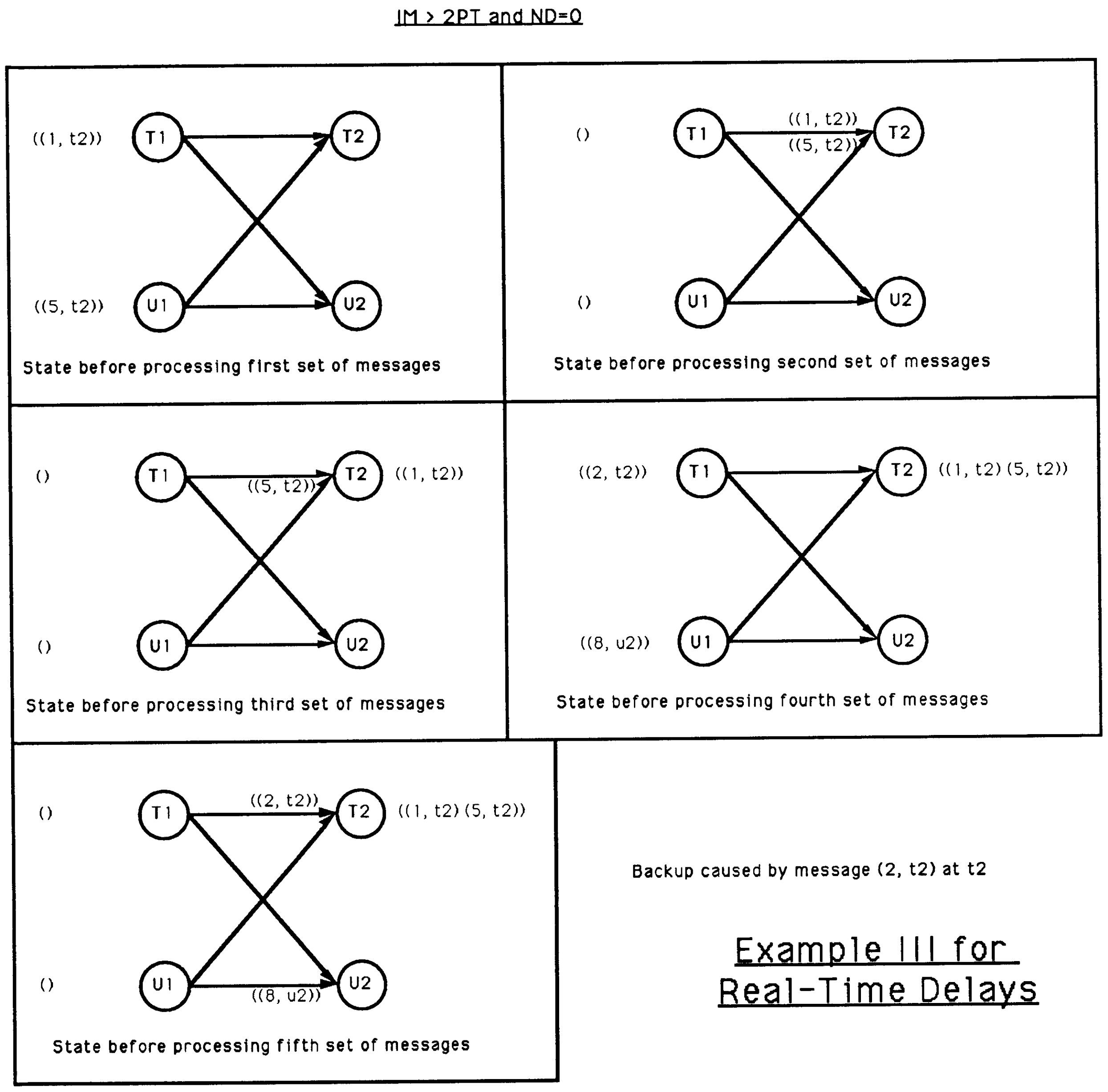

IM > 2PT and ND=0
((1, t2))
T1
T2
((5, t2))
U1
U2
State before processing first set of messages
()
((1, t2))
((5, t2))
State before processing second set of messages
((1, t2))
State before processing third set of messages
((2, t2))
((1, t2) (5, t2))
((8, u2))
State before processing fourth set of messages
State before processing fifth set of messages
Backup caused by message (2, t2) at t2
Example III for Real-Time Delays

Figure 7.28 Horizontal Partitioning Example III

7.7.2.3 $\frac{IM}{PT} > 2$

Whenever $\frac{IM}{PT} > 2$, then it is possible that either t_1 or u_1's message queues could be empty after each processor processes two tasks because processors process messages faster than twice the rate that they arrive. Backup is even more likely than in the second case. Figure **7.28** illustrates an example where backup occurs and $\frac{IM}{PT} > 2$.

Since task t_2 gets ahead in virtual time, backup will occur in this example. This is because the processor for partition t must process task t_2's second message at VRT=5 since t_1 will still have an empty message queue even after the processor processes the first message at t_2 with VRT=1. Since one partition is generally ahead of the other in virtual time, this phenomenon can always occur whenever ND is small. It is interesting to note that this example will not back up when $\frac{IM}{PT} < 2$, but examples I and II will back up when $\frac{IM}{PT} > 2$!

Without loss of generality, assume that partition t has a lower LVRT than partition u. Then as IM increases with respect to a fixed PT, t_2 is more likely to get ahead in virtual time (due to messages from u_1), because t_1's input queue is more likely to be empty after the processor at t finishes processing a message. As the IM increases the chance that t's processor will process more than one successive message at t_2 will grow. This is bad, because the more successive messages that are processed at t_2 before another is processed at t_1, the more likely it will be for a new message at t_1 to have a lower VRT than a message just processed at t_2 thus resulting in backup. This is due to the fact that if messages are processed faster than they arrive, t_1 will not be synchronized in virtual time with t_2.

The effect of $\frac{IM}{PT}$ makes the system act very much like the Network Paradigm simulation system. In our system, if $\frac{IM}{PT}$ is low, it is unlikely that queues will be empty. This is the safe condition of the Network Paradigm. When the ratio is high, a queue is likely to be empty, corresponding to the unsafe condition of the Network Paradigm.

7.8. Message Queue Length

7.8.1 Theory

There is an interesting relationship between the number of backups and the input message queue length at a task. Longer message queues were found to be associated with fewer backups, since a large message backlog allows messages in the input message queue to be sorted in virtual time order. Backups, on the other hand, cause longer message queues. When backup occurs, the backed-up task reprocesses messages that were previously processed by placing processed messages back on the task's input message queue.

Let us try to model this effect of message queue length. Given a task with message queue length n, what is the probability that an incoming message will cause a backup? Assuming the message queue is sorted in virtual time order, an incoming message will cause backup only if its virtual time is less than that of all n messages on the queue and if the first message on the queue is being processed. In other words the probability of an incoming message causing backup is bounded above by the probability that the incoming message has a virtual time less than that of all n messages on the queue.

We now estimate this probability. First we know that the virtual times of messages on a task's message queue are derived from independent identically distributed Poisson probability density function (pdf) with the same mean interarrival times (MITs). Then if all of the $n + 1$ messages arrive at approximately the same real time, then the probability that any particular one of those messages has the lowest LVRT is $\frac{1}{n+1}$. However, all the messages do not arrive at the same real time; in fact the messages already on the message queue have already arrived. We know that messages with larger RRTs (real receive times) probably have larger VRT. Therefore the probability that an incoming message will have a VRT less than those on the input message queue is bounded above by $\frac{1}{n+1}$. Similarly the probability of backup caused by an incoming message is bounded above by the same probability $\frac{1}{n+1}$. Therefore as n, the message queue length, increases, the probability of backup is likely to decrease.

We have quantitatively and qualitatively discussed relationships between backups and message queue lengths. At this point we would like to point out how our system is analogous to the Network Paradigm method. Unlike the Network Paradigm, it is possible for the system to be unsafe with non-empty queues. However, when message queues are long, our system approaches the safe category of the Network Paradigm system. As the size of the message queue decreases, the system becomes more unsafe.

In fact, the most unsafe message queue length for our system is zero which corresponds to the Network Paradigm's unsafe case of empty input message queues.

7.8.2 Experiments

Random Partitioning with 8 Partitions

Initial Processor Allocation	(4 4 4 4 4 4 4 4)
Effects of Increasing Time Delay	Message queue length decreases and number of backups increases
Scheme Ordering Based on # of Backups (ordering from worst to best)	1) Partitioning 2) Fixed-list 3) Circular-list and TQ 4) LVRT 5) EVRT
Lazy Message Cancellation's Effects on # of Backups	Reduced the number of backups dramatically. Most profound as real-time delay increased

Figure 7.29 Real-Time Delay for Longer Message Queue Lengths

As in previous experiments, all scheduling policies as well as lazy and aggressive message cancellation are tested. The effects of lengthening of the message queue were studied on a eight-input-node butterfly network, as opposed to the 16-input-node network studied in Section 7.5-7.7. We used the smaller network because the simulation run times were more manageable with a smaller network (given the fact that we were already adding real-time delays) and Concert was more prone to failure as the simulation run time grew. The experiments confirm our theories of an interesting relationship between the message queue length and the number of backups.

We developed two methods that artificially decrease queue lengths. They are:

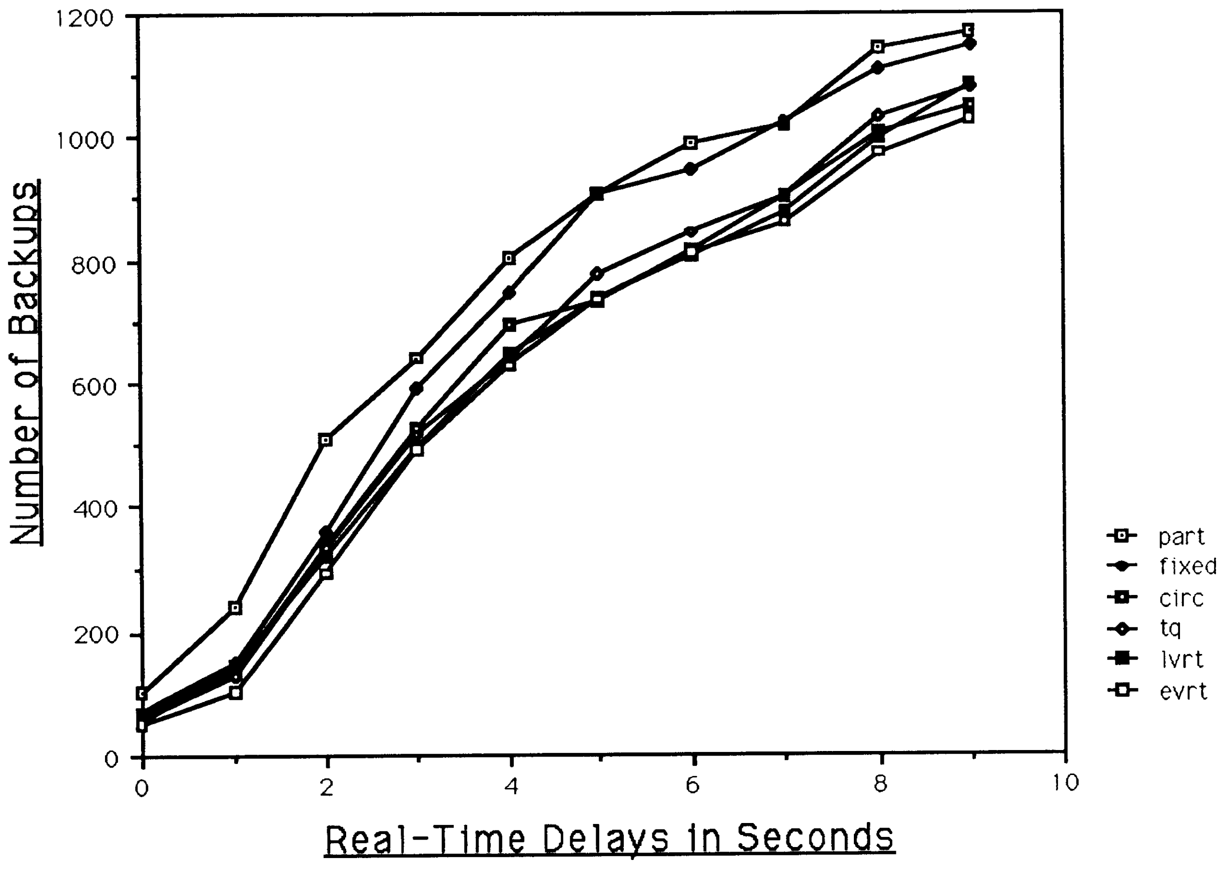


Figure 7.30 Number of Backups versus Real-Time Delay for r_8

1. Adding real-time delays between successive input messages.
2. Reducing the number of original input messages and adding a feedback loop in our network to maintain the same total number of messages into the input nodes.

The first experiment was used to confirm the hypothesis that real-time delays decrease the message queue length and increase the number of backups. In these experiments an eight-input-node butterfly network was used along with a random partitioning r_8 using eight partitions (note this experiment was done with an eight-input-node butterfly network instead of a 16-input-node butterfly network). Each of the eight partitions starts with four processors. The MIT of messages was fixed at 60 time units with the destinations of the messages random. The first of these experiments sent 50 messages into each of the eight input nodes at various VRTs, but with real time delays

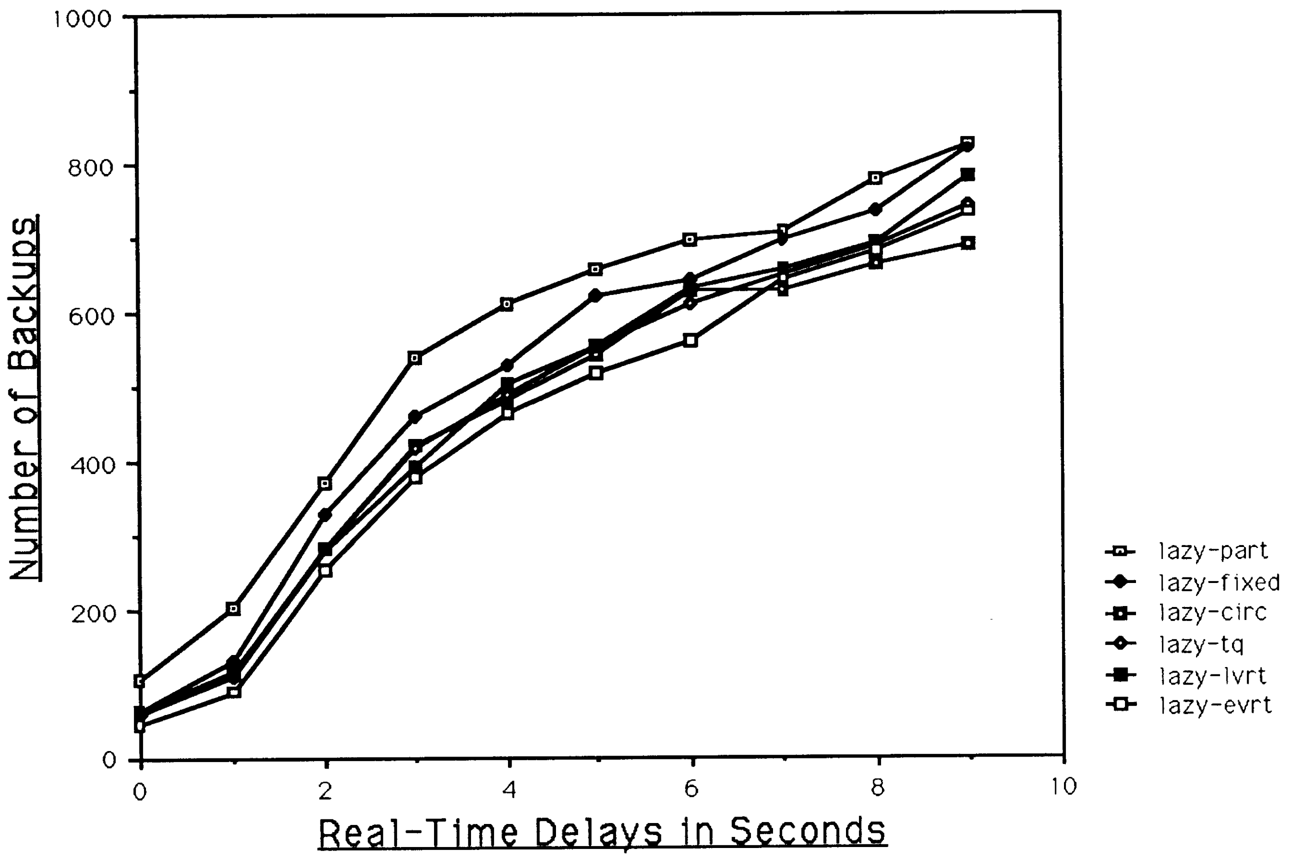


Figure 7.31 Number of Backups versus Real-Time Delay for Lazy Cancellation for r_8

between the messages ranging from 0 to 9 seconds.

The second experiment was used to show that reducing the number of original input messages and adding a feedback loop to maintain the same total number of messages into the input nodes also decreases the message queue length and increases the number of backups. In this experiment, only eight messages are sent into each of the eight input nodes at various VRTs. If a message arrives at its destination at a virtual time within 940 time units of the start of the simulation, then it is sent, after a virtual time delay, back to its originating input node to be resent through the network. This feedback effect also causes the message queue length to decrease because the average real-time delay between messages is increased.

The strategies that the experiments used were: static partitioning, fixed-list,

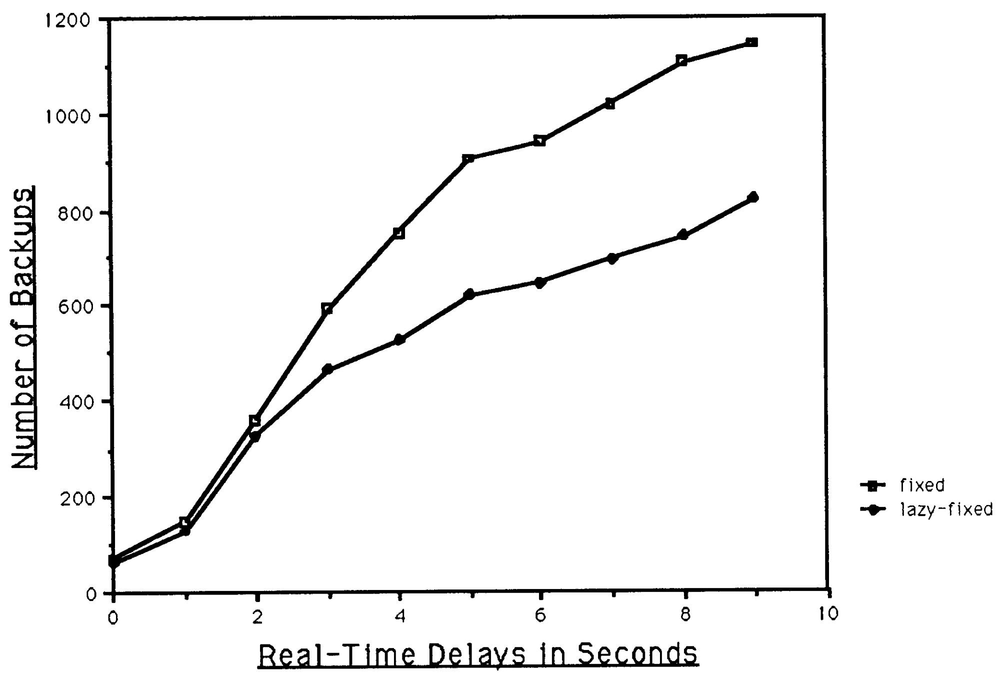


Figure 7.32 Number of Backups versus Real-Time Delay using Aggressive or Lazy Cancellation with Fixed-list scheduling for r_8

circular-list, longest task queue, lowest LVRT, and lowest EVRT dynamic repartitioning. Each experiment compared lazy and aggressive message cancellation, with at least two trials for each message cancellation policy.

7.8.2.1 Real Time Delay

The experiment is summarized in Figure **7.29**, while results of this experiment are shown in Figure **7.30** with the real time delays in units of seconds. The results for zero real time delay will correspond to those experiments (no real-time delay) performed in the previous sections except they do not use the same topology. Figure **7.30** shows the number of backups for each of the scheduling policies as the real time delay increases. As the real time delay increases, the number of backups increases. The static partitioning scheme performed the worst (most backups) followed by the fixed-list, whereas the EVRT scheme performed the best.

Lazy message cancellation reduced the number of backups dramatically for all the

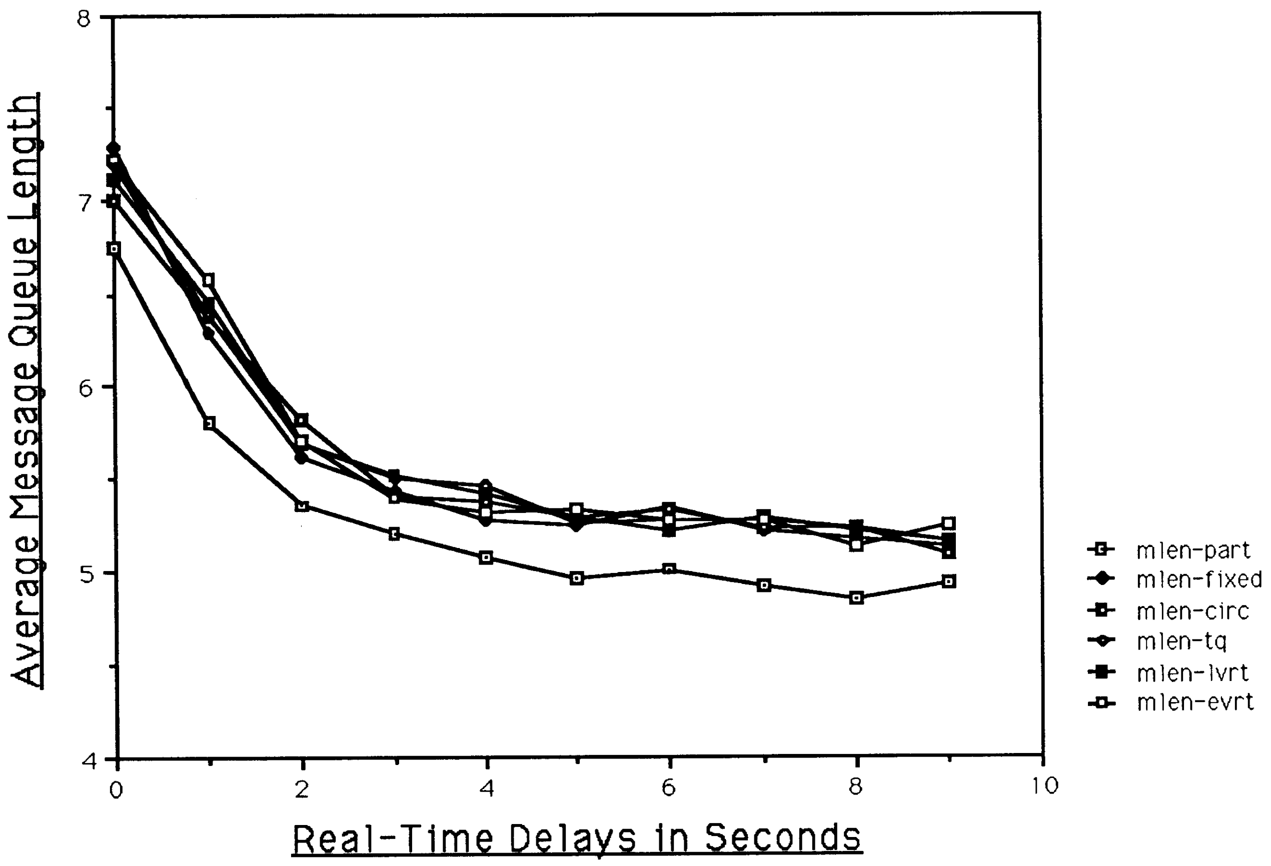


Figure 7.33 Message Queue Length vs Real-Time Delay for r_8

scheduling schemes. In particular lazy message cancellation proved to be more effective as the real-time delay between messages increased. This effect is shown for the fixed-list policy in Figure **7.32**.

Processing times of the different schemes were recorded. However, the results are not shown because added real-time delays increased the processing times of the simulations accordingly.

Figures **7.33** and **7.34** plot the average message queue length M versus real time delays for the aggressive and lazy message cancellation cases. M in Figures **7.33** -**7.36** is defined to be the average over all tasks T of M_T, the average message queue length at T. M_T is computed by recording the real time and the input message queue length every time a new message is added to or removed from the queue. From these data, we

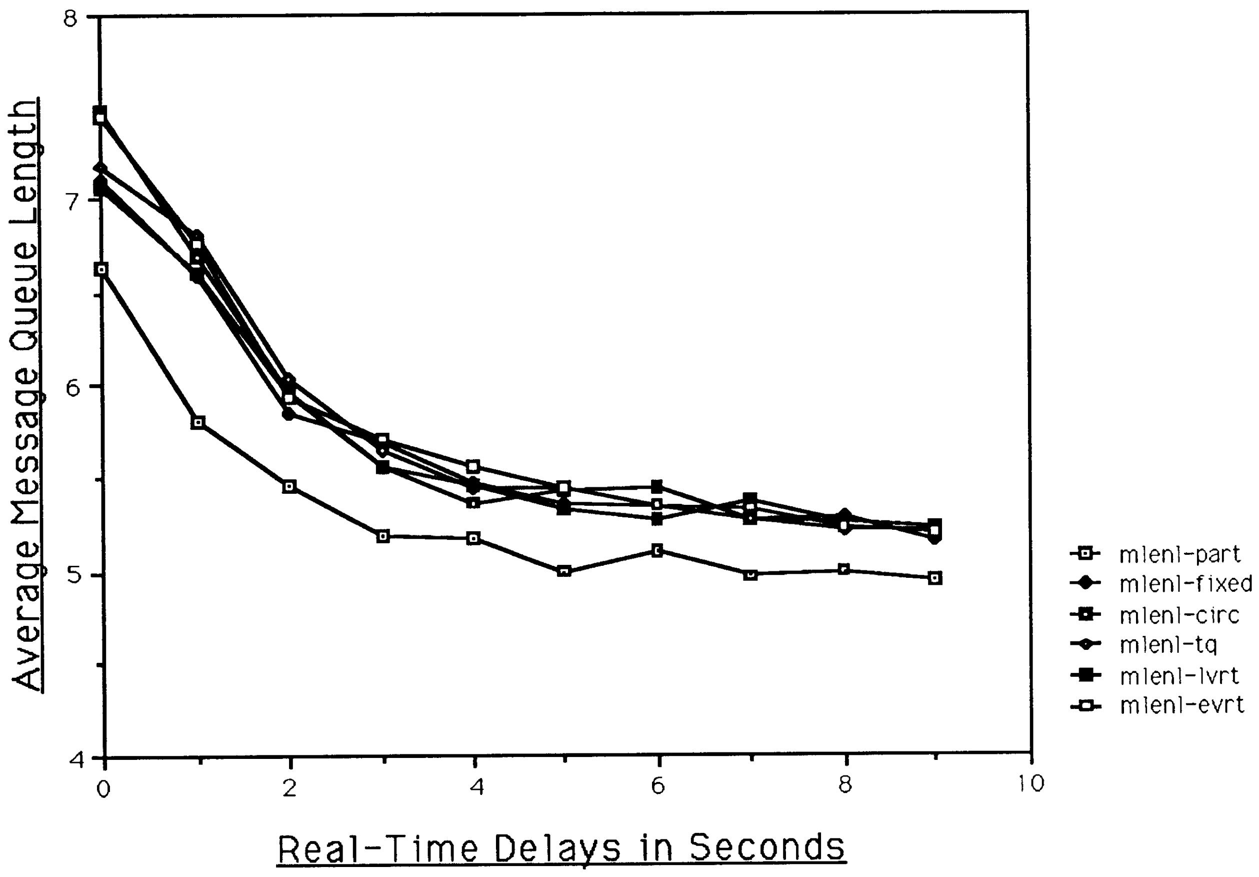


Figure 7.34 Message Queue Length vs Real-Time Delay for Lazy Cancellation r_8

can derive the duration of each interval when the input message queue length of any task is L. Therefore,

$$M_T = \sum_L \frac{L \cdot (Duration\ of\ L)}{Duration\ of\ L}$$

The policies are static partitioning (mlen-part, mlenl-part), fixed list (mlen-fixed, mlenl-fixed), circular list (mlen-circ, mlenl-circ), lowest task queue (mlen-tq, mlenl-tq), lowest LVRT (mlen-lvrt, mlenl-lvrt), and lowest EVRT (mlen-evrt, mlenl-evrt). As the time delays increase, the message queue length decreases for each scheduling policy. The sharpest dropoff in message queue length is between 0 and 1 second of real time delay.

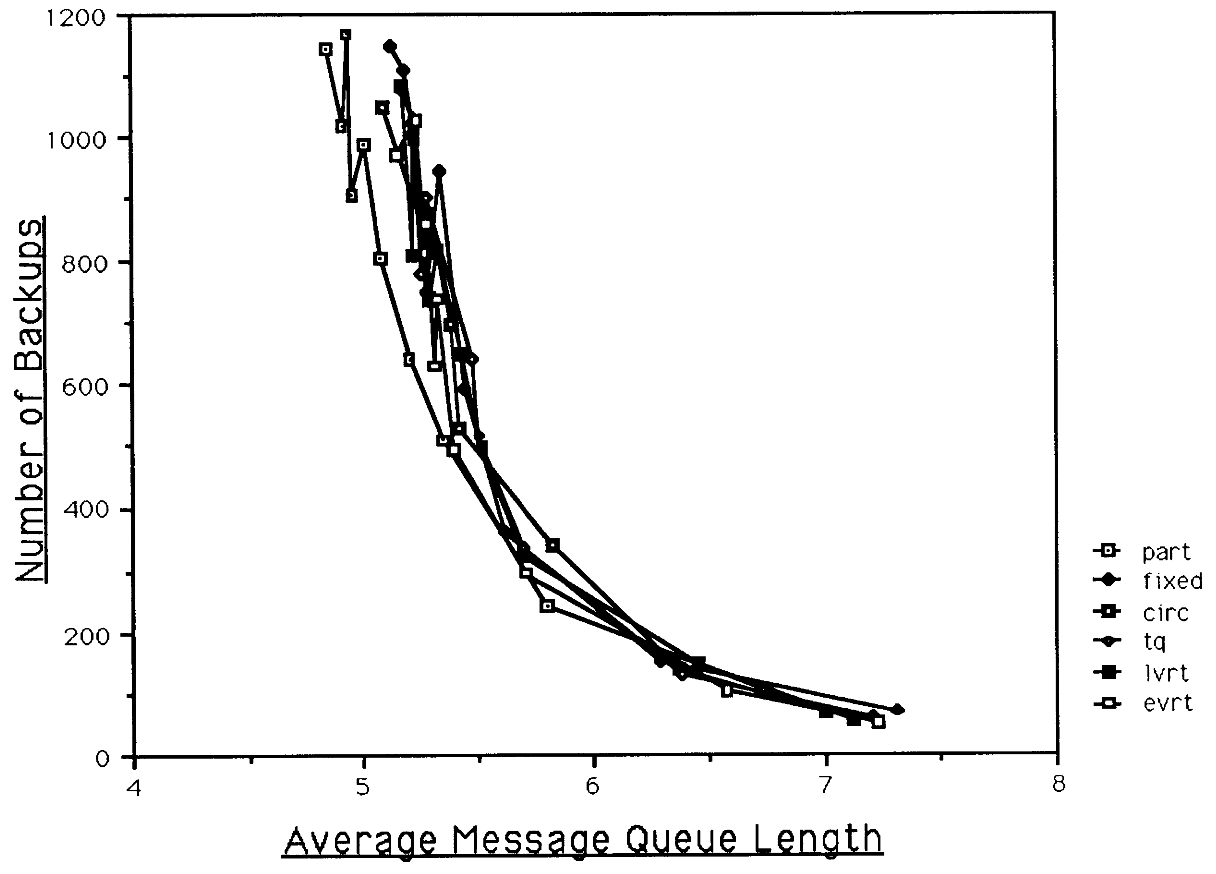


Figure 7.35 Number of Backups vs Average Message Queue Length for Aggressive Message Cancellation for r_8

Figure **7.35** shows the number of backups for different message queue lengths for each of the different scheduling policies with their lazy counterparts shown in Figure **7.36**. This is another view of the data, plotting backups versus the average message queue length M, instead of against real-time delays. Figures **7.35** and **7.36** show that the number of backups increase dramatically as the message queue length approaches 5. For message queue lengths between 6 and 7 the slope of the curve for backups is not as steep.

One may ask why the message queue lengths are always greater than four, even when there are real-time delays of up to 9 seconds. If IM is very long, each new message will cause a flurry of backups and recomputation, which should all die down before the

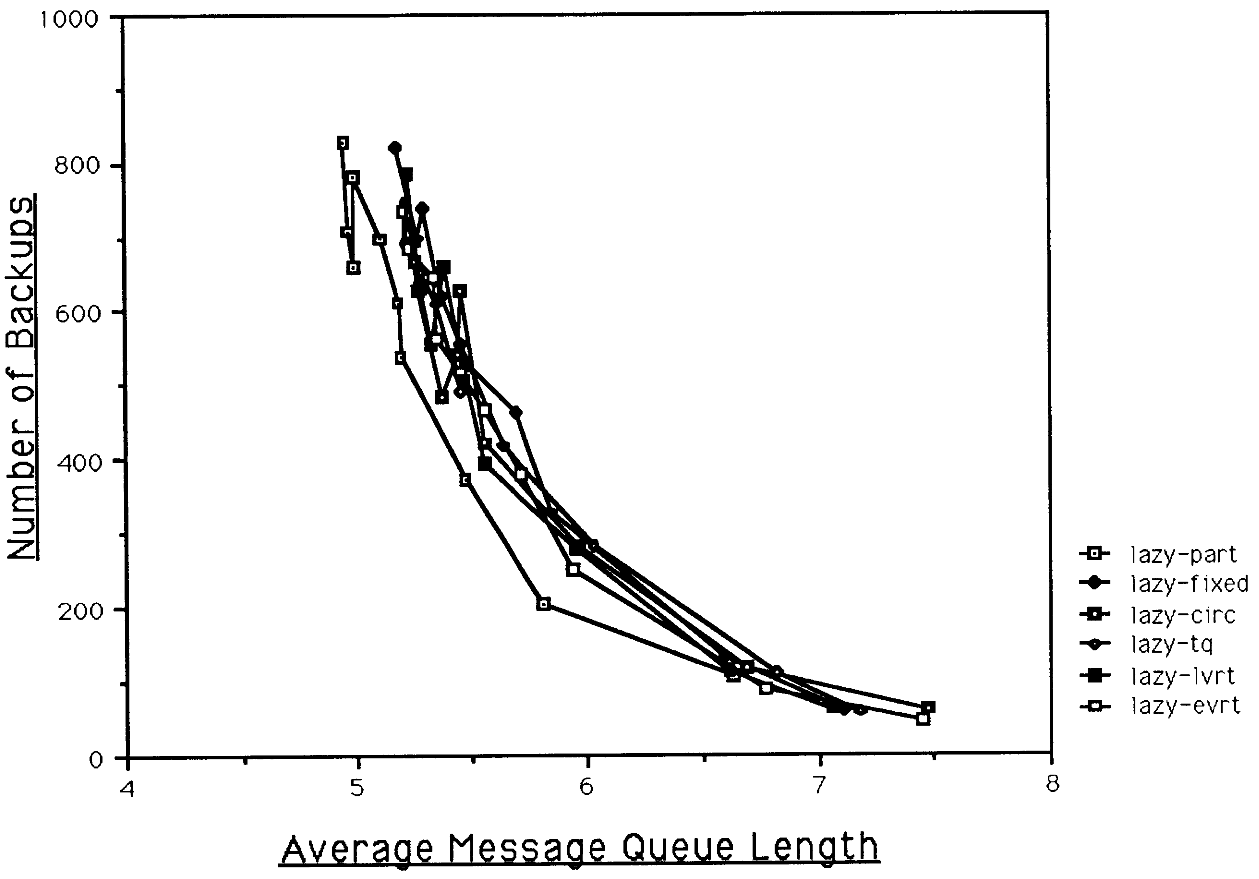


Figure 7.36 Number of Backups vs Average Message Queue Length for Lazy Message cancellation for r_8

next input message arrives, leaving all input message queues empty. Thus, if we average over time, average message queue length should approach 0 as IM approaches infinity, without any further increase in backups after IM exceeds a critical value. However, in this example shown in Figure **7.30** it is clear that the number of backups is still increasing when IM=9; therefore, the critical value of IM has not been reached. Now, whenever a backup occurs at a task, all of the messages that need to be reprocessed are replaced on the task's input message queue. Therefore, since the number of backups is extremely high it is understandable that the average message queue lengths at any node could be higher than four.

7.8.2.2 Feedback Loop

This experiment used the same eight-input-node network with random partitioning, and the same input messages as the previous experiment; however, only eight of the 50 messages were sent into each of the eight input nodes. If a message arrived at its destination probe before 940 virtual time units, it was sent, after a virtual loop time delay, back to the originating input node to be resent through the network.

The feedback effect caused the message queue lengths to decrease because fewer messages were initially sent to the input nodes. The messages did not return to the input nodes until the messages traversed the system (via the feedback loop). This increased the average real-time delay between successive messages. The real-time delay decreased the message queue lengths in the same manner as discussed in the previous section.

Figure **7.37** shows the results of this experiment. It plots the number of backups for each scheduling scheme while using the feedback network. Since each scheduling scheme was executed twice for each message cancellation policy, two points are plotted for each scheduling scheme under the aggressive message cancellation policy and two points for each scheduling scheme under the lazy message cancellation policy. The number of backups for the aggressive message cancellation ranged from 390-540 which was many times larger than that of the 50-110 backups of the zero time-delay shown in **7.30** (in the previous experiment). Lazy message cancellation reduced the number of backups substantially in all of the scheduling schemes. The average message queue lengths for each of the scheduling policies varied from 1.05 to 1.2, which is much lower than the message queue lengths for any other experiment.

7.8.2.3 Horizontal versus Vertical Partitioning

We also compared the message-queue lengths for horizontal partitioning H_8 to those for vertical partitioning V_6 in the 16-input-node butterfly network (no feedback). These results are illustrated in Figures **7.38**, and **7.39**. The policies are static partitioning (mq-part), fixed list (mq-fixed), circular list (mq-circ), longest task queue

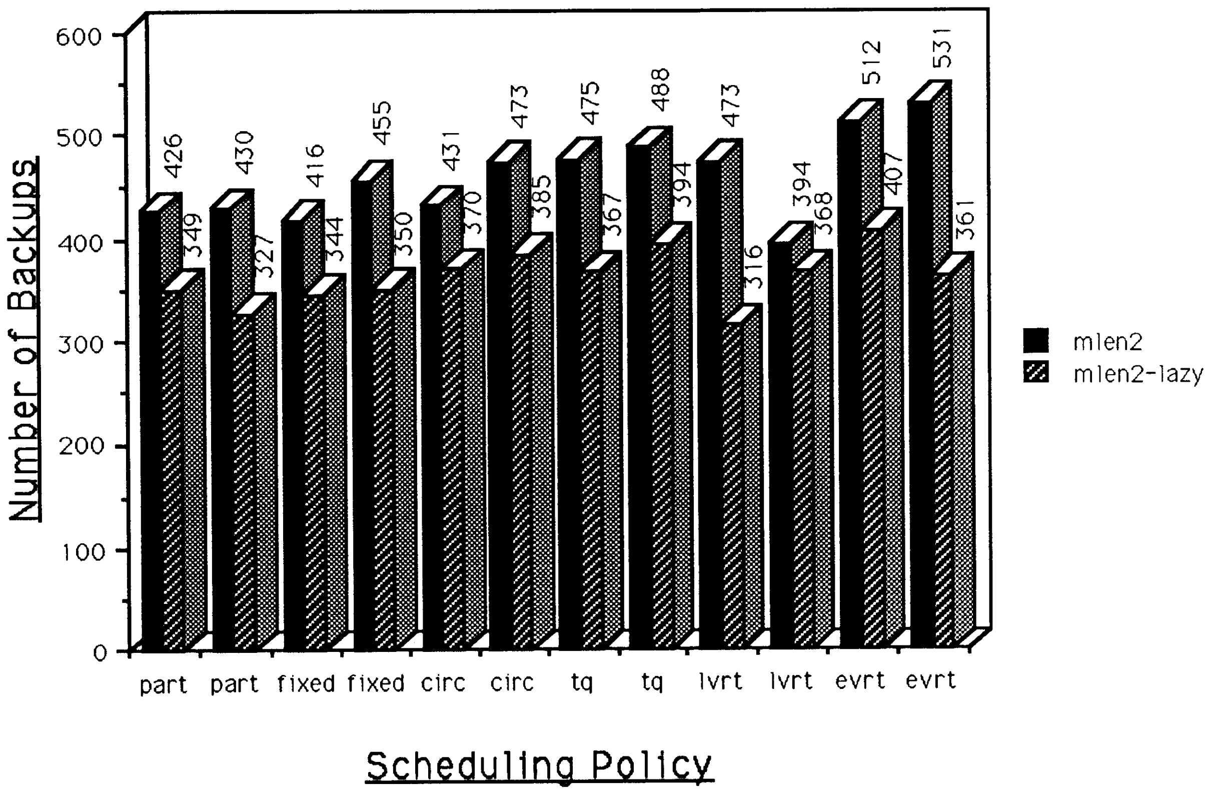


Figure 7.37 Number of Backups versus Scheduling Schemes for Feedback-Loop for r_8

(mq-tq), lowest LVRT (mq-lvrt), and lowest EVRT (mq-evrt). Note that there is essentially no trend in the message queue lengths as MIT increases. The message queue lengths in horizontal partitioning tended to be lower than those of vertical partitioning. Our results showed that horizontal partitioning had less than half the message queue lengths of vertical partitioning.

7.8.3 Discussion

The goal of this section is to establish that decreasing the message queue length at each task will also increase the number of backups. Our results confirm this hypothesis. Our results also show that we can control the average message queue length by altering

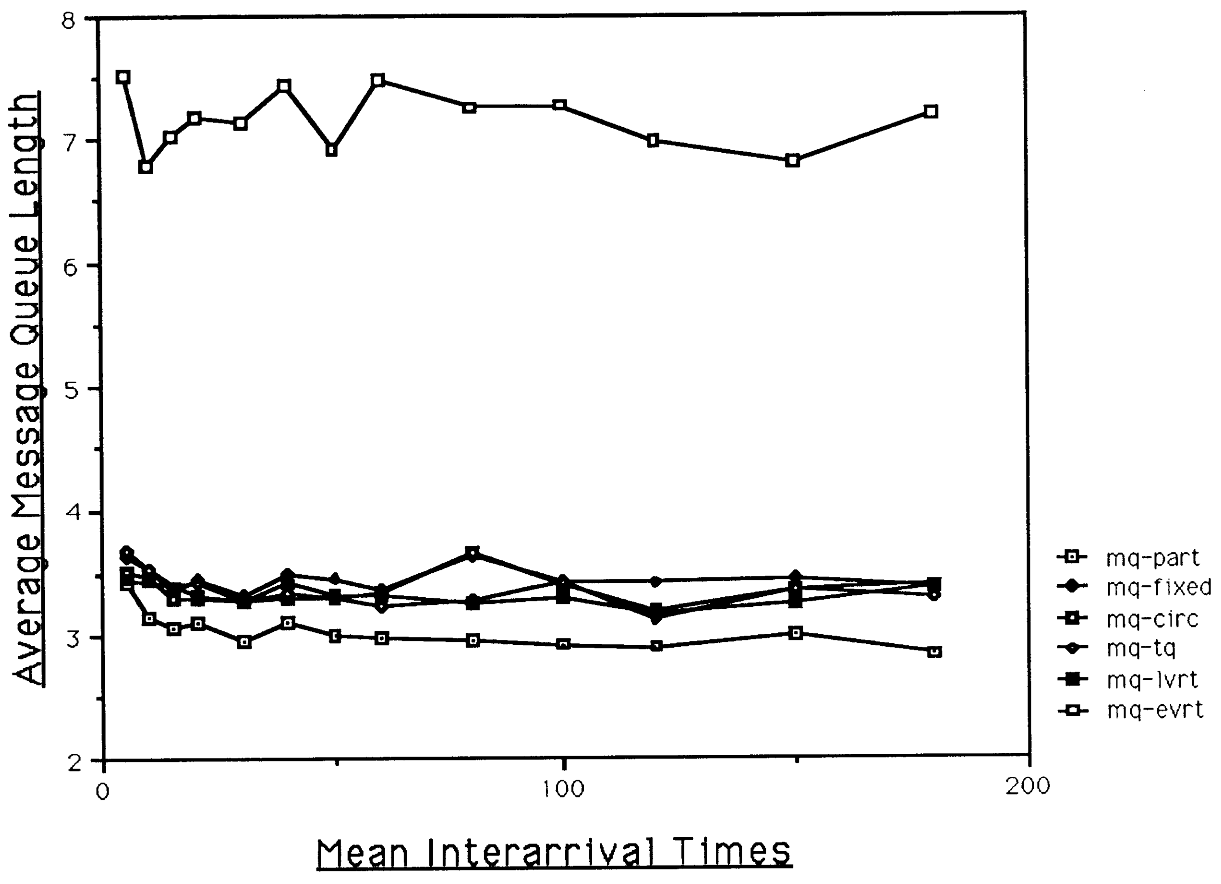


Figure 7.38 Average Message Queue Lengths for H_8

time delays and by adding a feedback loop.

The message queue lengths in horizontal partitioning tended to be much lower than those in vertical partitioning. In an earlier section we showed that the horizontal partitioning method had more backups than that of vertical partitioning. This increase in backups was caused by the aggressive backup effect and the synchronization effect in horizontal partitioning. In addition to more backups, we showed that horizontal partitioning had less than half the message queue lengths of vertical partitioning. This shows that decreasing the average message queue lengths at each task increases backups.

Our results showed that there is almost no trend in the message queue lengths as MIT increases. However, the number of backups increases greatly as MIT increases for horizontal partitioning (see Figure 7.7). We attribute this to the fact that a small

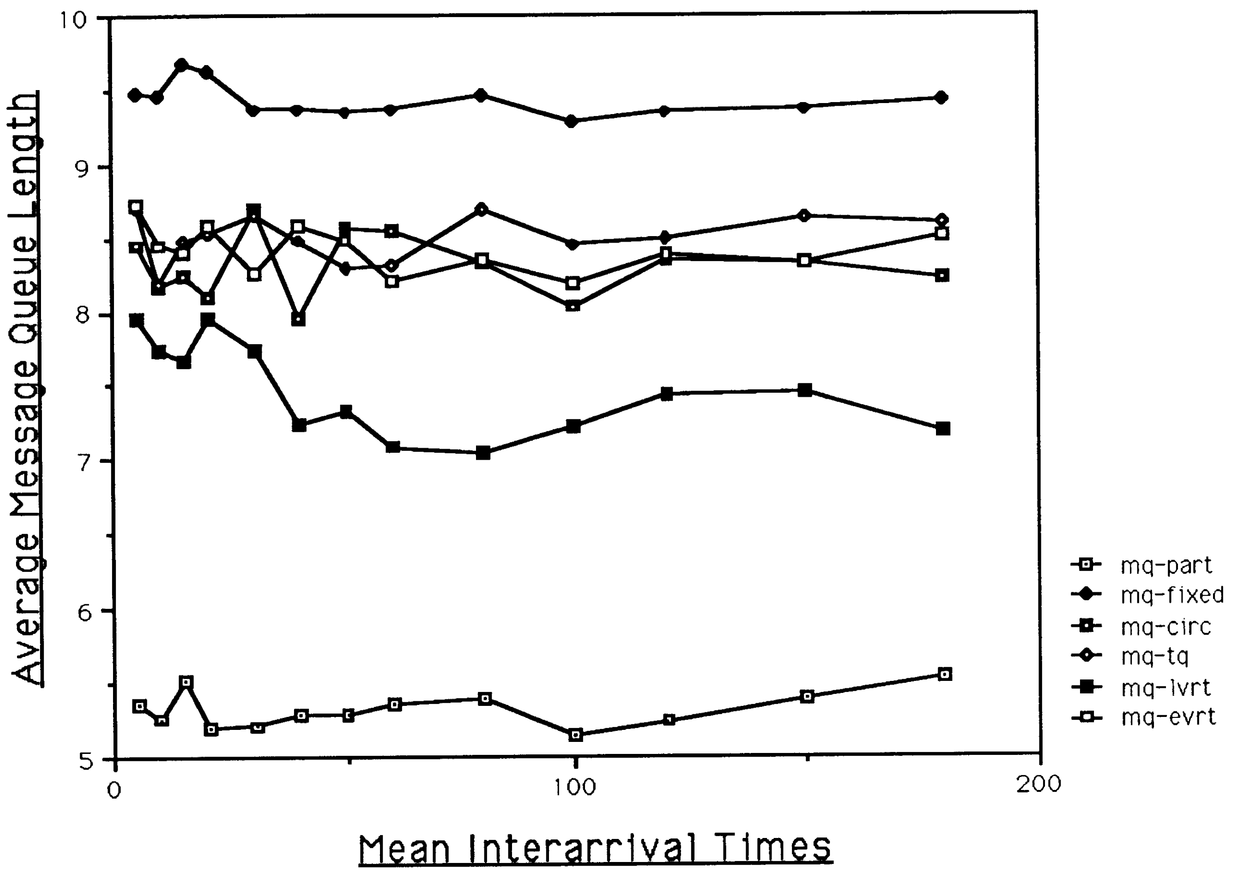


Figure 7.39 Average Message Queue Lengths for V_6

decrease in queue length may greatly increase the number of backups. For example, in Figures **7.38** and **7.39**, there is a small initial decrease in the message queue lengths. This corresponds to a large initial increase in the number of backups in Figure **7.7**. This coincides with Figure **7.35** which shows that a very small message queue length differential potentially causes a very large change in the number of backups.

Since the experiments r_8 and H_8 are very similar, it is plausible that results shown for r_8 in Figure **7.35** would be very similar if H_8 were used.

It is interesting to note that lazy message cancellation increased the message queue length for all of the scheduling schemes. By increasing the message queue length, lazy message cancellation in turn reduced the number of backups.

7.8.3.1 Application to Other Simulations

Message queue lengths and backups have an interesting relationship. We have shown quantitatively and qualitatively that shorter message queue lengths can cause more backups. On the other hand more backups tend to increase the message queue lengths; because when tasks back up, those messages that have already been processed and have greater virtual time than the preempting message are placed back on the input message queue of the task. Therefore, if the average message queue length is small, then the simulation will have many backups. This large number of backups will increase the message queue length, which in turns decreases the backups later in the simulation. Therefore, for each simulation there is an equilibrium point that is reached where the average message queue length remains constant. This equilibrium effect explains why the message queue lengths of Figure **7.38** is relatively flat for different MITs.

If this generalization is true, one may ask why horizontal partitioning has much lower message queue lengths than vertical partitioning shown in Figures **7.38**, and **7.39**. Let us try to determine why this is true. In horizontal partitioning with dynamic scheduling, processors traverse the network from higher precedence classes to lower precedence classes along with the messages they process. Therefore, messages are more likely to queue up at the input nodes (where backups do not occur) than at other nodes.

However, in vertical partitioning (using dynamic scheduling), processors prefer to process input nodes before processing other nodes because all the processors are initially assigned to the input partition and there is an abundance of work in the input partition. Even if messages become available in other stages, the processors will prefer to continue processing nodes in the input partition because a processor moves only if there is no work in its partition. Therefore, messages will queue up at nodes other than input nodes. This explains why *only* non-input nodes should have smaller message queues in horizontal partitioning. However, the message queue length at non-input nodes is the important parameter because backups only occur at non-input nodes and we are

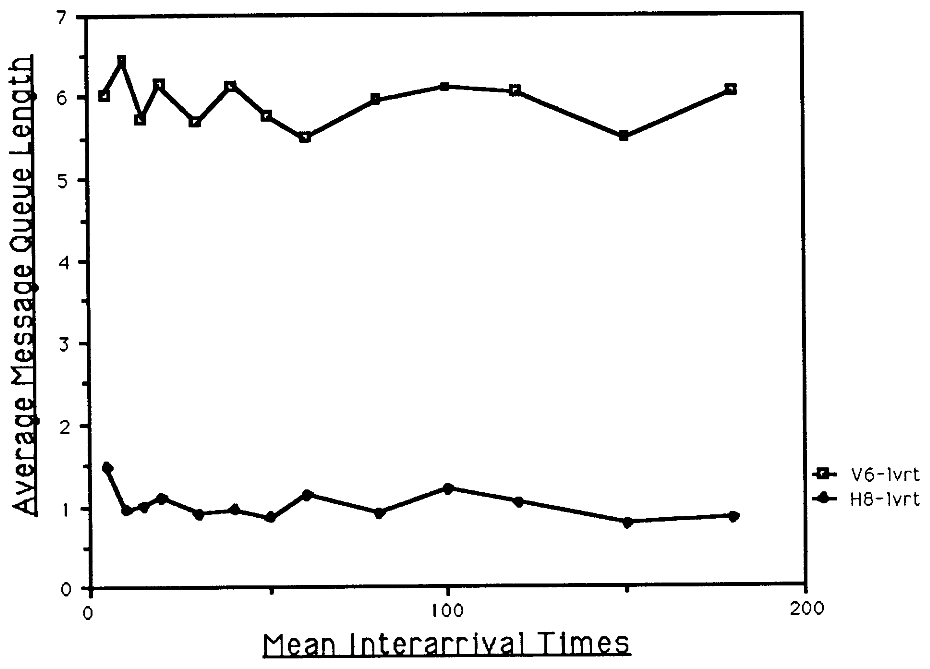


Figure 7.40 Average Message Queue Lengths for H_8 and V_6 for Non-Input Nodes only

interested in the relationship between backups and message queue lengths. Figure 7.40 shows the average message queue length for H_8 (H_8-lvrt) and V_6 (V_6-lvrt) at the non-input nodes for the LVRT scheme. This confirms that horizontal partitioning has lower message queue lengths than vertical partitioning at the non-input nodes.

The effects of varying real-time delays and virtual-time delays are analogous. Recall the generalizations made under virtual-time delays in Section 7.5.3.3:

1. When ND is high compared to MIT, there is implicit vertical partitioning because the processors tend to process the messages in the input partition first.
2. When ND is low compared to MIT, more backups occur than when ND is high because processors find input nodes less attractive to process.

Compare this to what happens under real-time delays:

1. When PT is high compared to IM and ND > 0, there is implicit vertical partitioning due to ND, and high PT will ensure non-empty message queues at the input nodes.

This will allow for processing of messages at the input nodes first which is good.

2. When PT is low compared to IM, more backups occur than when PT is high because processors are forced to process nodes other than the input nodes due to their empty message queues.

The three sources of backups are:

1. Real or Virtual Time delays (discussed immediately above),
2. Race Conditions, and
3. Partition Saturation (to be discussed in Section **7.10.4**).

We give an example of where race conditions cause backup when real-time delays and virtual time delays are not significant (i.e., ND=10,000 and $\frac{PT}{IM} > 1$). For example in V_6, suppose that two tasks a_1 and a_2 in the input partition p_1 have messages m_1 and m_2 respectively, (somewhere on their input queues) destined to some task T in the partition p_2 as shown in Figure **7.41**. Let's say due to the abundance of processors, tasks a_1, a_2, and T all have processors processing them. Suppose that:

1. m_2's VRT $<$ m_1's VRT,
2. m_1 gets processed at a_1 and is sent to T,
3. m_1 is finished processing at T before m_2 is finished processing at a_2.

Then backup will occur at T, due to race conditions.

In this section we are only concerned about the first cause of backups. In order to simplify our analysis, for the rest of this section we look at network simulations with static partitioning and only one processor per partition. By restricting one processor per partition while using static partitioning, we can assure that each partition will only have one processor for the duration of the experiment. Partition saturation does not occur when there is only one processor per partition. Some race conditions, such as the one shown above, can be avoided by restricting one processor per partition. On the other hand, some race conditions, such as the aggressive backup effect, will still occur.

We examine the effect of varying real-time delays when virtual-time delays are not significant (i.e., ND=10,000). Similarly, we examine the effect of varying virtual-time

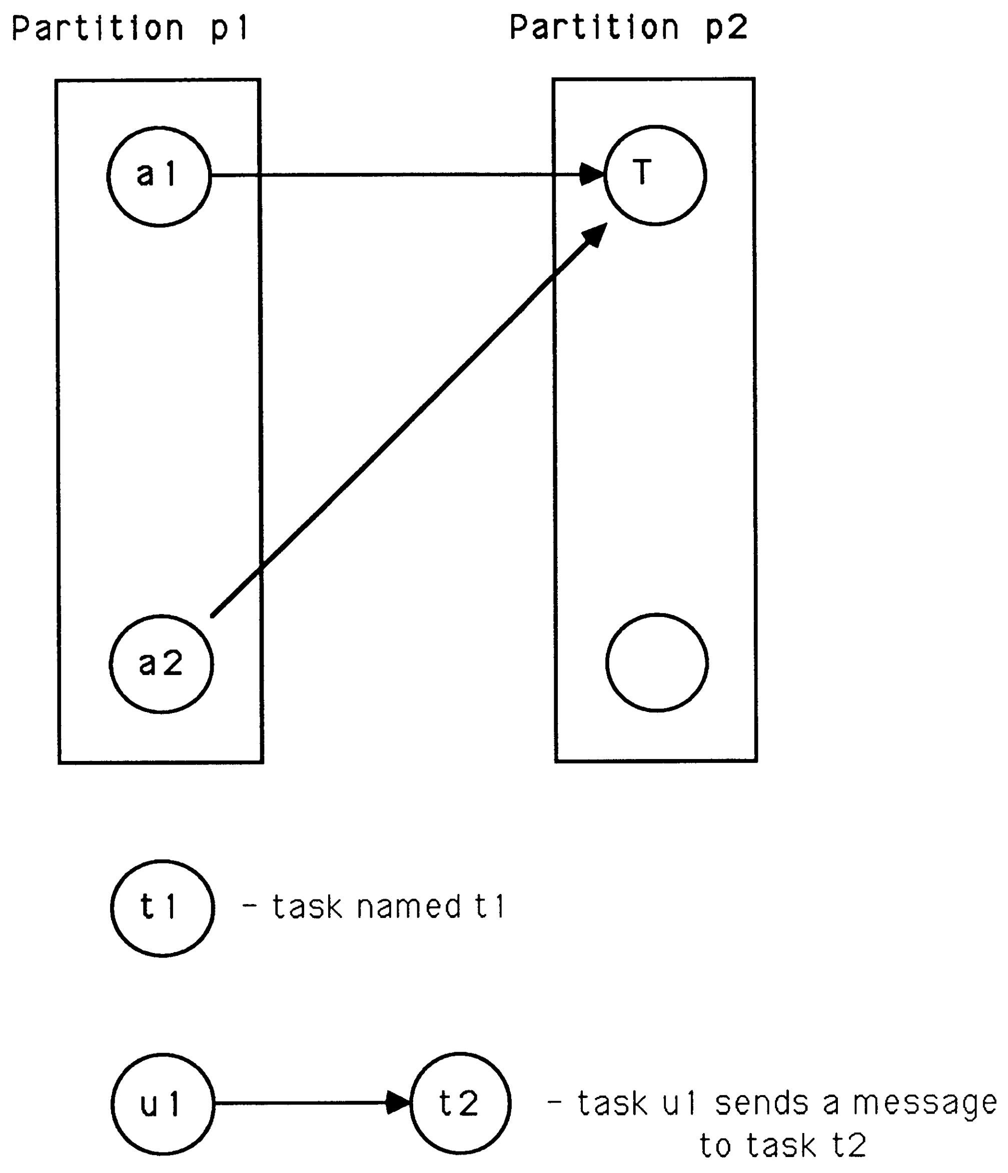


Figure 7.41 Example of a Race condition with no real or virtual-time delays

delays when real-time delays are not significant (i.e., $\frac{IM}{PT} < 1$). A summary of these effects are shown is Figure **7.42**. In this figure real-time delays are abbreviated "RTD" whereas virtual-time delays are abbreviated "VTD". The column under the low virtual-time delays is supported by the results performed on our four-node network model of Section **7.2**. Since it is very difficult to vary real-time delays on Concert (due to its inherent delays), we performed these experiments on our four-node network model on

Static Partitioning with 1 processor per Partition!				
Partitioning Method	Low VTD, High ND vs MIT ND=10,000, MIT=5 Justified by Expt. run on Our 4-node Model		Low RTD, High PT vs IM PT=1, IM=0 Justified by Expt. run on Concert	
	Low RTD High PT vs IM	High RTD Low PT vs IM	Low VTD High ND vs MIT	High VTD Low ND vs MIT
Vertical Partitioning	Prevents Backups	Many Backups Regardless of VTD	Prevents Backups	Prevents Backups Vertical Part is immune to VTD
Horizontal Partitioning	Forces Vertical Part Prevents Backups	Many Backups	Forces Vertical Part H_8had no Backups H_{16}had 0 or 1 Backup	Many Backups

Figure 7.42 Real and Virtual Time Delay effects for different Partitioning Methods

a Symbolics Lisp Machine keeping track of real time in software. These results (for ND=10000) are displayed in Figure **7.43**. The right hand column under the low real-time delays is supported by additional experiments run on Concert with varying ND and MIT which are shown in Figure **7.44**. Note that the experiment for low VTD and low RTD (for both vertical and horizontal partitioning) appears to be the same experiment. However, one experiment is done on Concert whereas the other experiment is done on a Lisp Machine.

Using our model, the graph showing the number of backups for vertical partitioning as the real-time delays increased remained the same for different levels of virtual-time delays (ND=0, 10, 100, 1000, 10000). This is shown in Figure **7.45**. This is consistent with our theory that virtual-time delays do not affect vertical partitioning. On the other hand, the graph for horizontal partitioning remained the same only for small

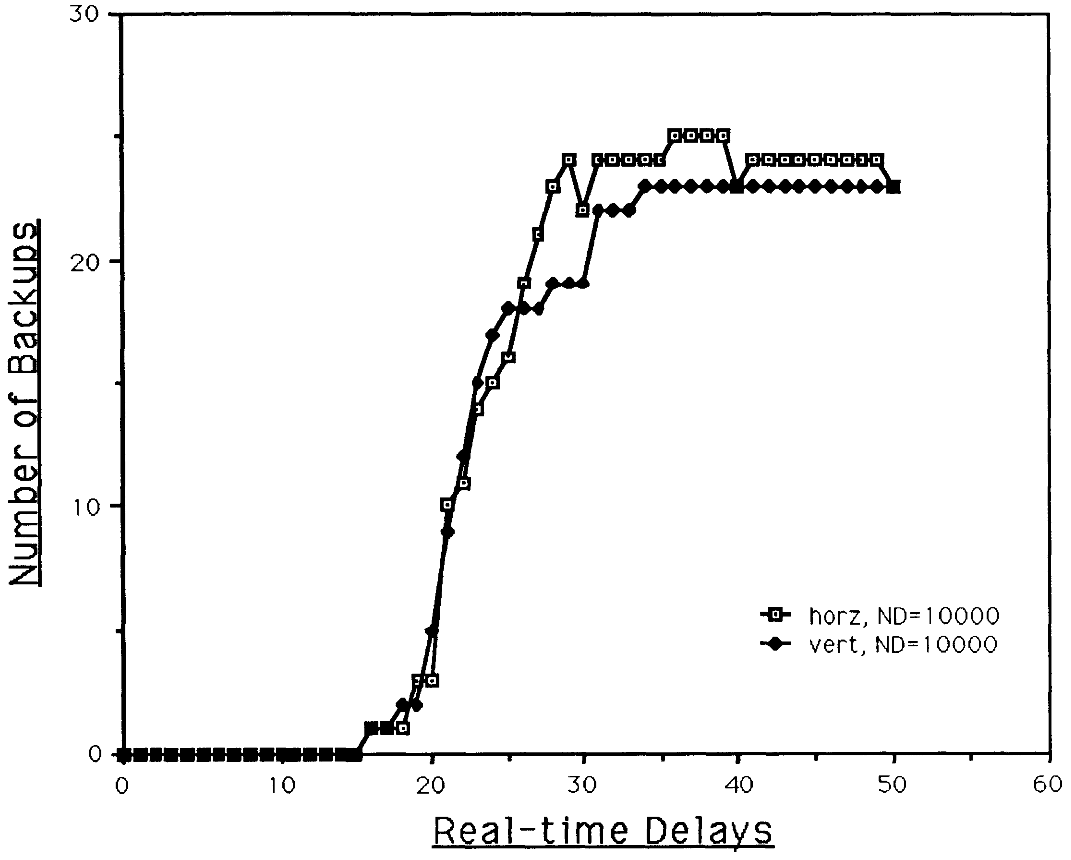


Figure 7.43 Real-Time Delay Effects with PT=10 with varying IM, with insignificant Virtual-Time Delays (ND=10000)

virtual-time delays (ND $\geq$ 100). This is shown in Figure **7.46**. We also examine the real-time delays effects when virtual-time delays are high (ND=0). This is shown in Figure **7.47**.

The analogy between real-time delays and virtual-time delays is not completely symmetrical. One difference is that virtual-time delays do not affect vertical partitioning. On the other hand, real-time delays always affect horizontal or vertical partitioning and virtual-time delays always affect horizontal partitioning. For example, when PT is low enough, real-time delays can cause backups in vertical partitioning no matter what the virtual-time delays.

Another difference is that the result of reducing real-time delay is not analogous to

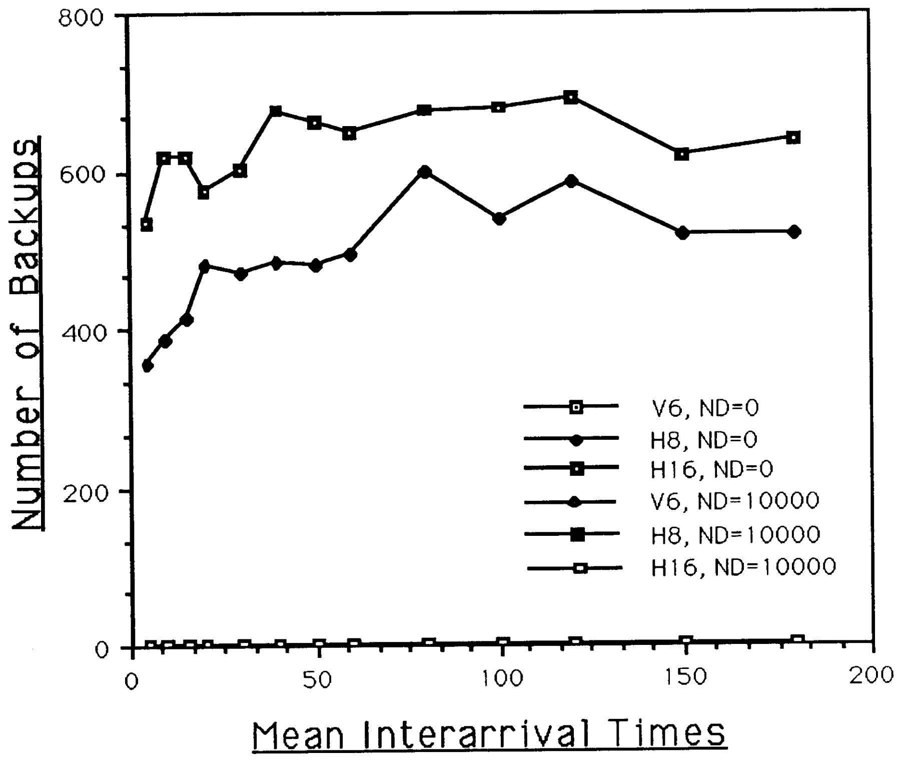


Figure 7.44 Backups for V_6, H_8, and H_{16} for ND=0 and 10000, with low Real Time Delays, Note that the plots for H_8 with ND=10000 and V_6 are level at zero

that of reducing virtual-time delay. With high real-time delays, reducing virtual-time delays does nothing to reduce backups. On the other hand, when virtual-time delays are high (ND=0 and 10), reducing real-time delays can eliminate backups from vertical partitioning but not from horizontal partitioning. This is shown in Figures 7.47 and 7.46.

Eliminating real-time delays is not sufficient to eliminate backups in horizontal partitioning. This is because backups caused by inter-partition messages will still occur in horizontal partitioning when there are virtual-time delays (ND $\leq$ 10). Although

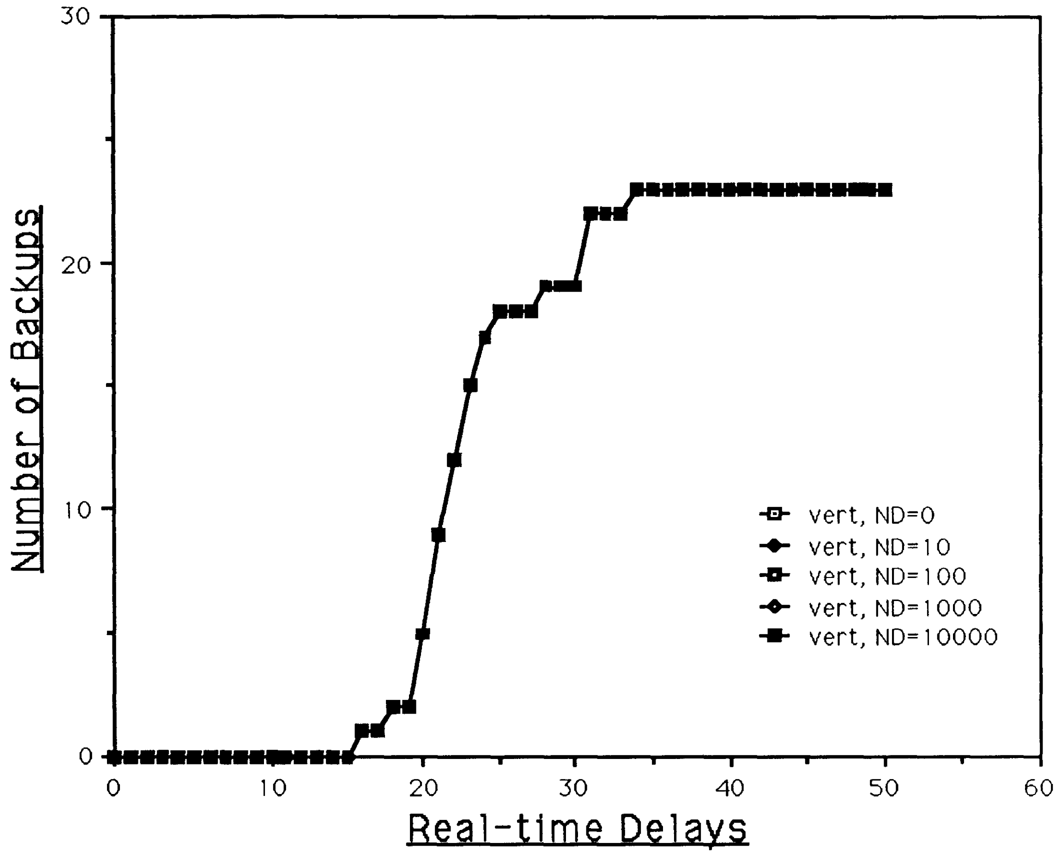


Figure 7.45 Real-Time Delay Effects, Comparing Vertical Partitioning with various Virtual-Time Delays (ND=0, 10, 100, 1000, 10000) Note that all of the plots for V_6 are the same regardless of ND

by raising PT, one can assure there will be messages in all the input nodes when a processor is deciding which message to process, one cannot guarantee, just by raising PT or lowering IM, that the input node(s) will be processed before all output nodes. On the other hand, by reducing virtual-time delays (ND $\geq$ 100), one can force the simulation to process all messages in the lower precedence classes before processing these inter-partition messages, thus preventing most backups (in the absence of real-time delays). This is because in horizontal partitioning the virtual-time delays govern how attractive messages at input nodes are to processors.

In general real-time delays can be present in any simulation system. Real-time

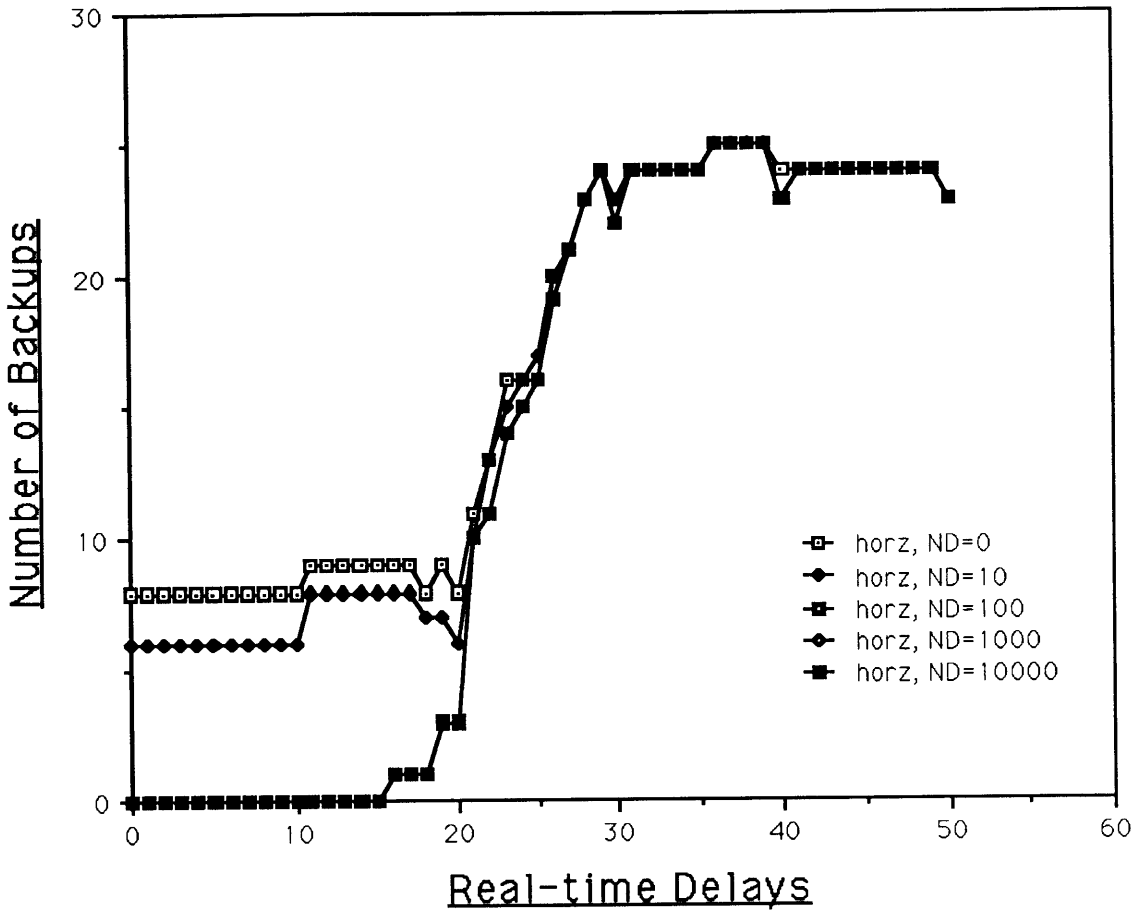


Figure 7.46 Real-Time Delay Effects, Comparing Horizontal Partitioning with various Virtual-Time Delays (ND=0, 10, 100, 1000, 10000)
Note that all the plots for ND=100, 1000 and 10000 are the same

delays between messages can be caused by three different phenomena:

1. Initial real-time delay between successive input messages. This occurs when the simulation depends on input messages that are results of *other* computations and these results slowly trickle out.
2. The hardware of the multi-processor used for the simulation. The actual real time a processor spends to processes a message for a particular task causes an implicit real-time delay between successive messages.
3. Feedback.

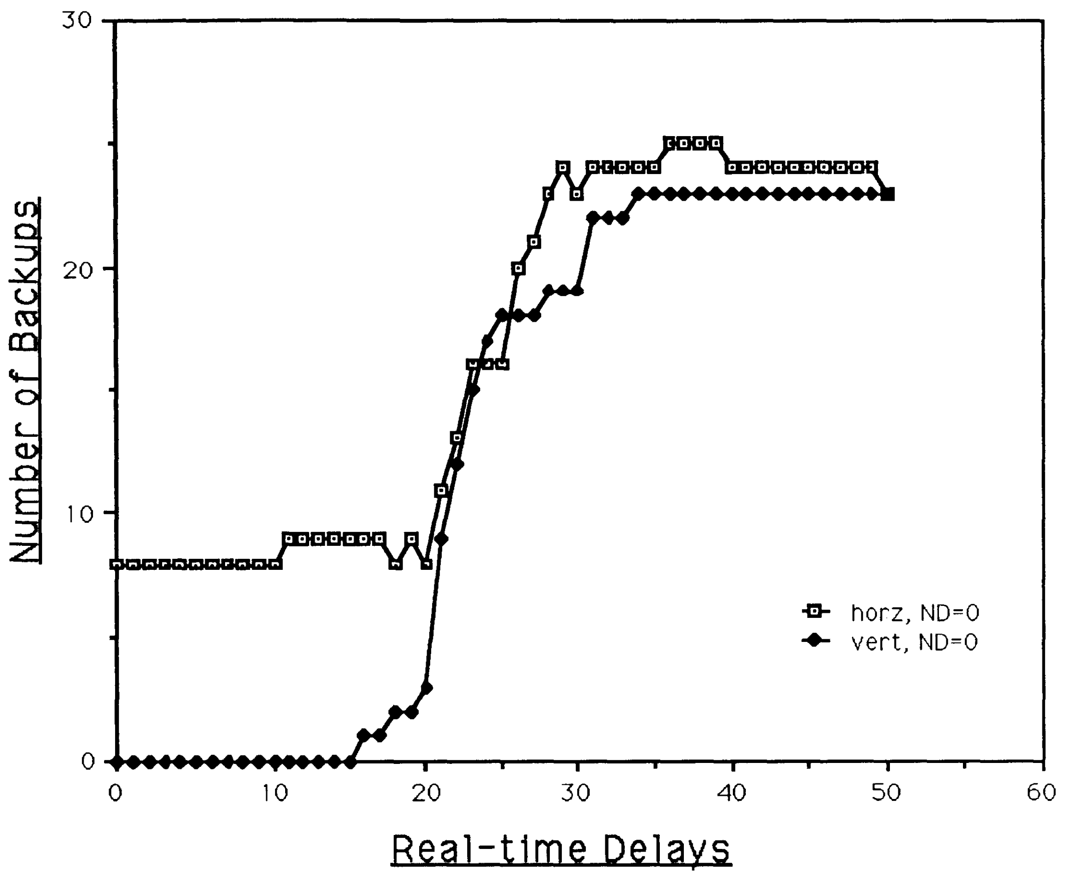


Figure 7.47 Real-Time Delay Effects with PT=10 with varying IM, with significant Virtual-Time Delays (ND=0)

The results suggest that as many of the input messages as are known should be placed into the system at initialization. This will reduce the real-time delays between successive messages at the beginning of the simulation. This can be done in the first problem area if we do the other computations first before starting our simulation; therefore, all of the input messages will be available when the simulation begins, thus eliminating initial real-time delays.

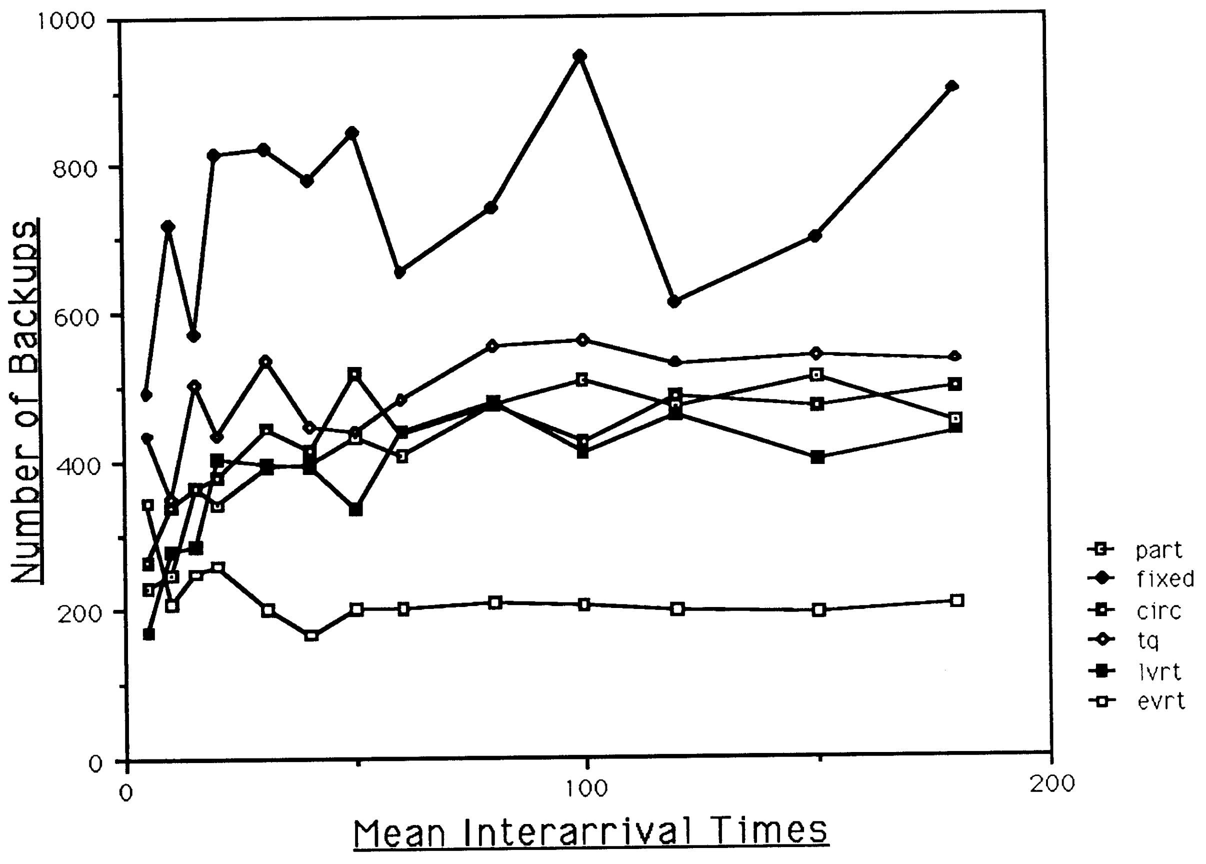


Figure 7.48 Number of Backups versus MIT for R_{8c}

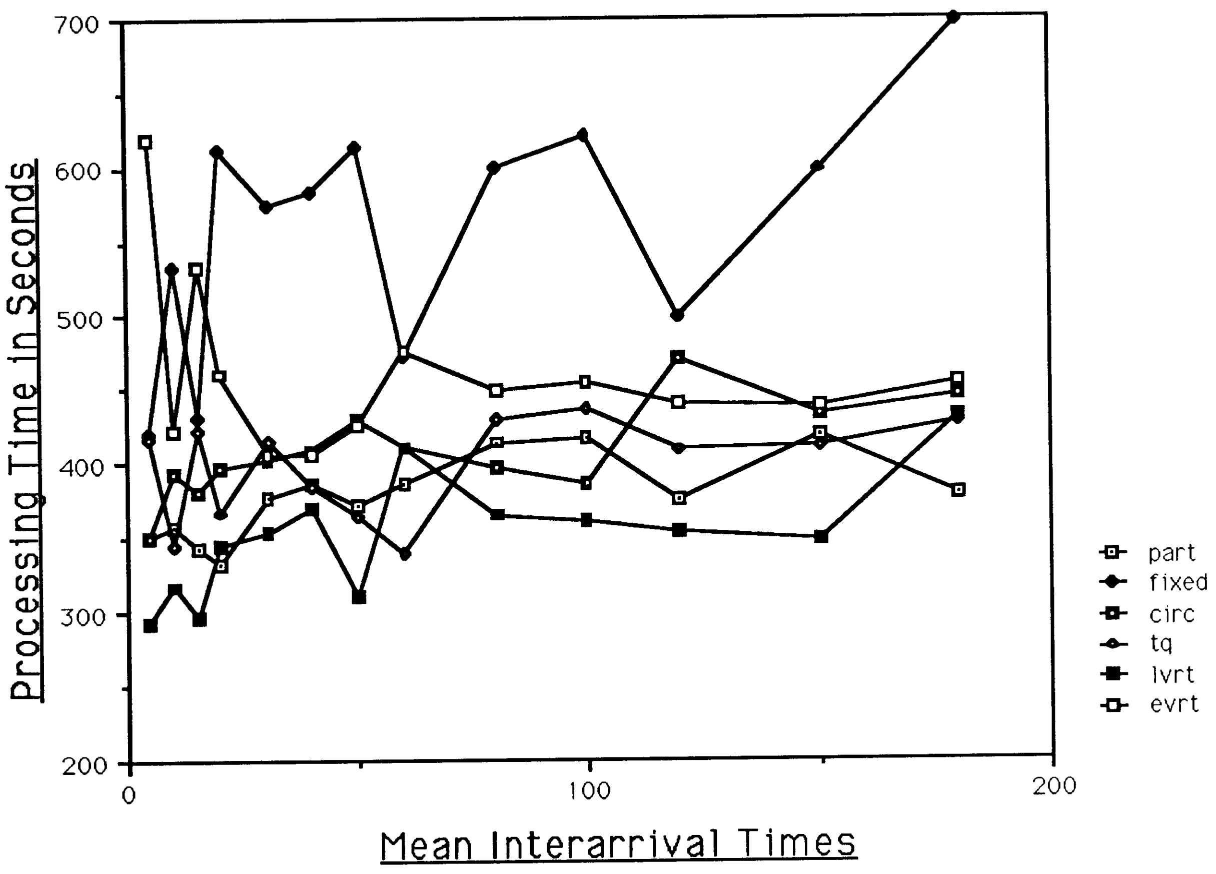


Figure 7.49 Processing Time versus MIT for R_{8c}

Random Partitioning with 8 Partitions

Initial Processor Allocation	(4 4 4 4 4 4 4 4)
Effects of Increasing MIT on # of Backups	# of Backups increases for all schemes except for EVRT where it decreases
Scheme Ordering Based on # of Backups (ordering from worst to best)	1)Fixed-list 2) TQ 3) Partitioning, Circular-list, and LVRT 4) EVRT
Lazy Message Cancellation's Effects on # of Backups	Reduced dramatically for all schemes
Effect of Increasing MIT on Processing Time	Processing time increases for all schemes except for EVRT where it decreases
Scheme Ordering Based on Processing Time (ordering from worst to best)	1) Fixed-list 2) EVRT 3) Partitioning, Circular-list and TQ 4) LVRT
Lazy Message Cancellation's Effects on Processing Times	No improvement

Figure 7.50 Summary of Random Partitioning III with 8 Partitions

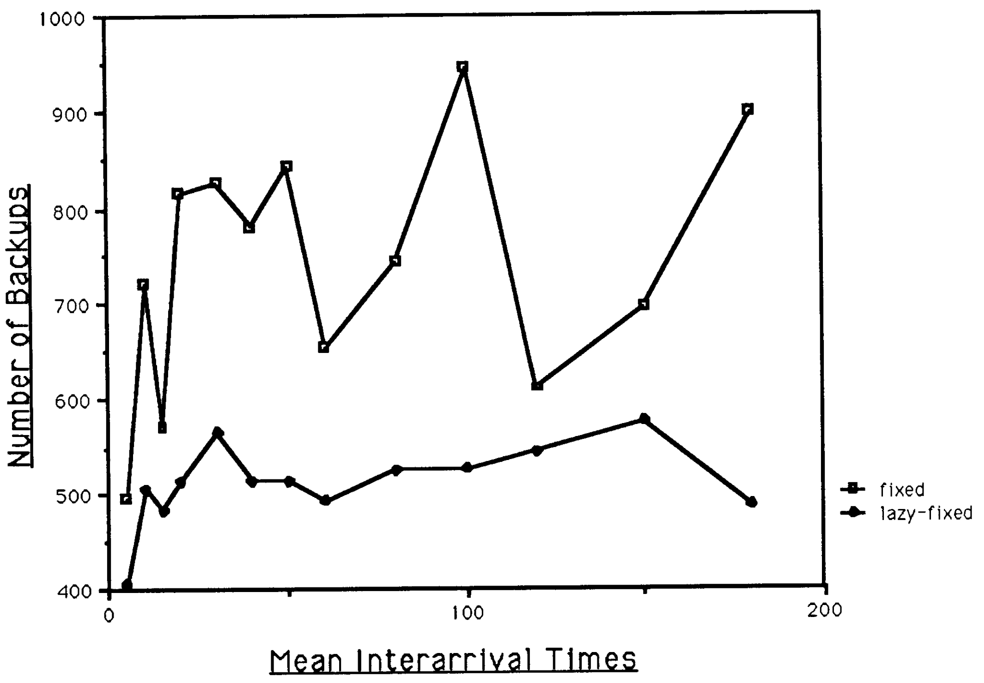


Figure 7.51 Number of Backups versus MIT for R_{8c} using Aggressive or Lazy Cancellation with Fixed-list scheduling

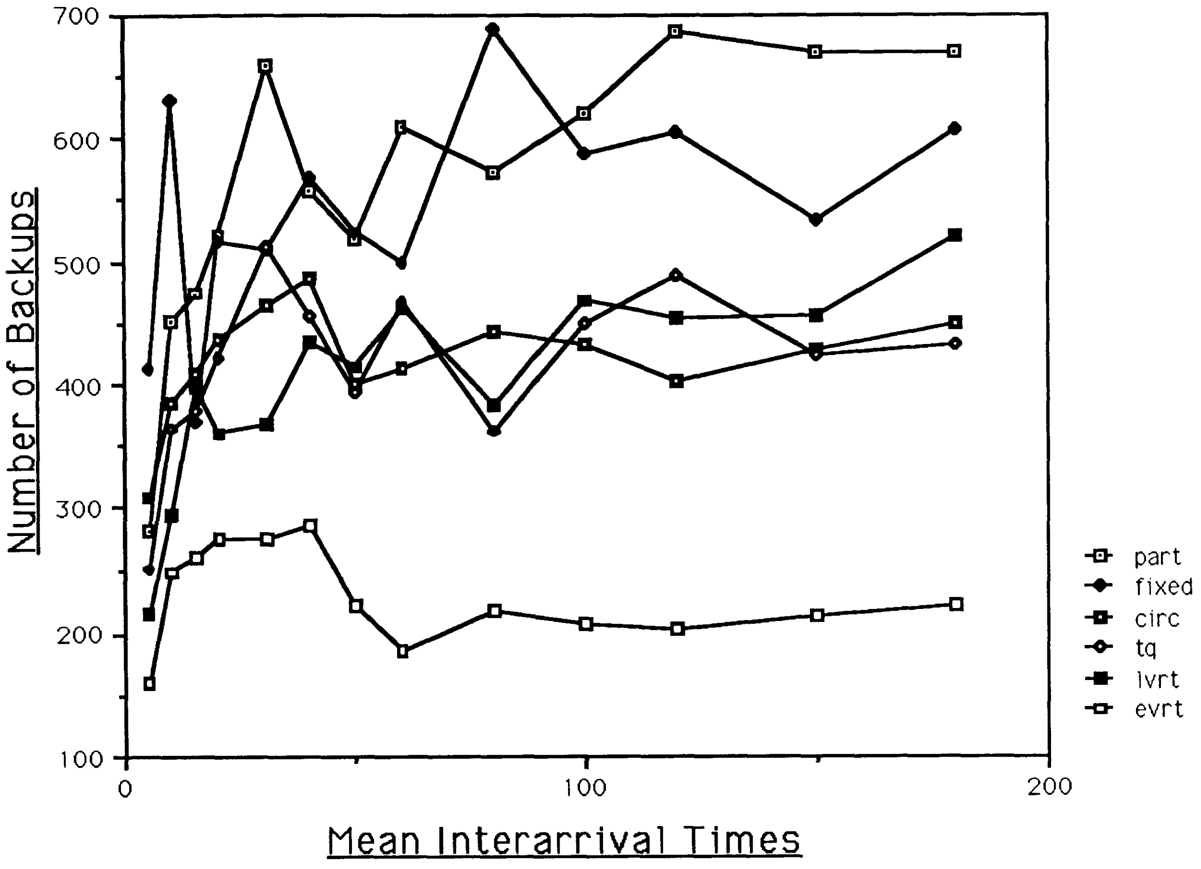


Figure 7.52 Number of Backups versus MIT for R_{16b}

7.9. Random Partitioning

7.9.1 8-Partition Case

The experiment was done with three different ways of randomly partitioning the 16-input-node butterfly network into eight partitions. The partition schemes are shown in the Appendix. Each partition was initially allocated four processors each since it was unclear what allocation would be optimal. The results of one experiment (using the R_{8c} partitioning – see Section **5.5.2**) is shown in Figures **7.48**, and **7.49**; the results of the others are similar. Figure **7.48** shows the number of backups for each of the scheduling policies as the MIT increases. Figure **7.49** displays the processing times for each of the schemes as the MIT increases.

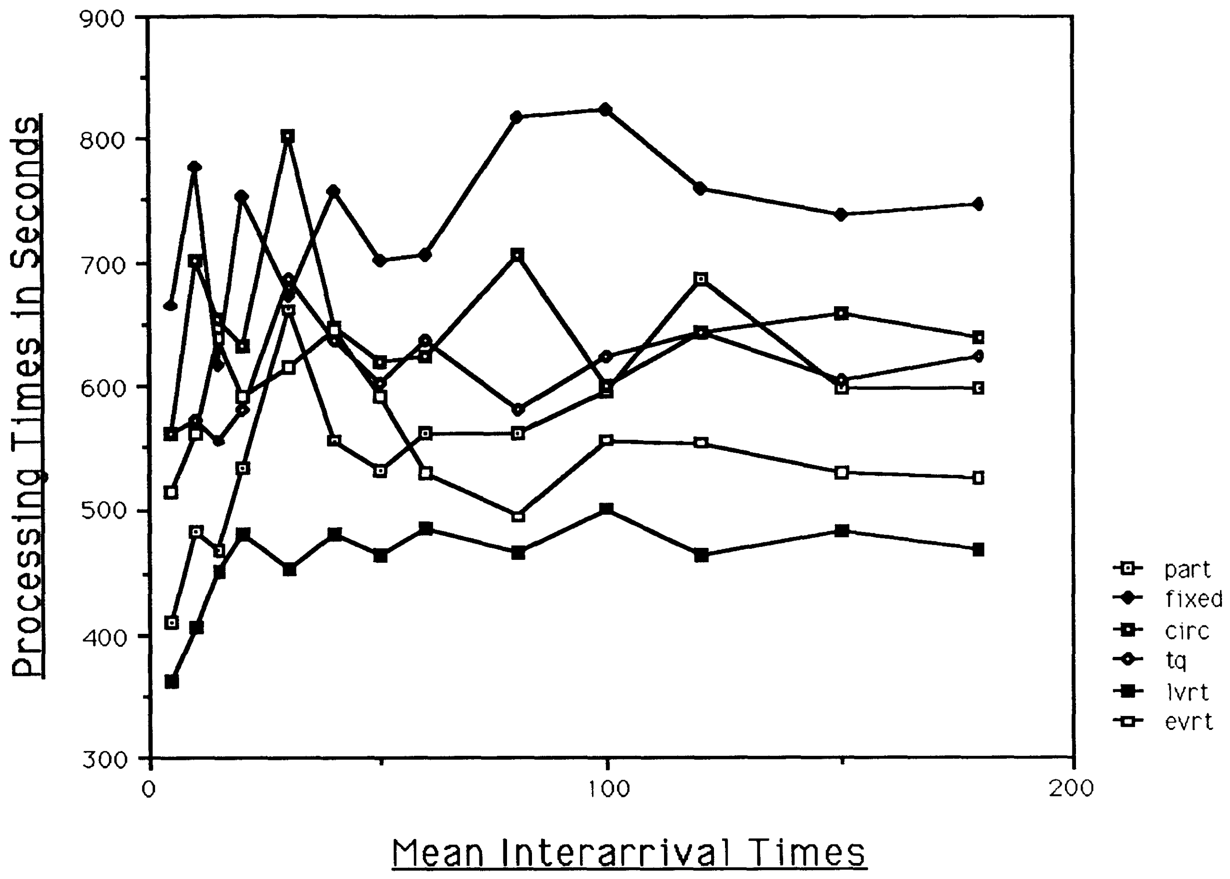


Figure 7.53 Processing Time versus MIT for R_{16b}

The EVRT scheme caused the fewest backups while the fixed-list scheme had the most. The LVRT scheme caused the lowest processing times whereas the fixed-list scheme had the longest processing times. Also, except for the EVRT scheme, all schemes performed worse in terms of backups and processing time as the MITs increased. However, the EVRT scheme maintained its performance even through increases in MITs. This is very similar to the horizontal partitioning case. The results of the R_{8c} experiment are summarized in Figure **7.50**.

Lazy message cancellation was almost always better than aggressive message cancellation in reducing the number of backups in random partitioning for all of the scheduling schemes. This effect is shown for the fixed-list strategy in Figure **7.51**.

Random Partitioning with 16 Partitions

Initial Processor Allocation	(2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2)
Effects of Increasing MIT on # of Backups	# of Backups increases for all schemes except for EVRT where it decreases
Scheme Ordering Based on # of Backups (ordering from worst to best)	1) Partitioning and Fixed-list 2) Circular-list, TQ, and LVRT 3) EVRT
Lazy Message Cancellation's Effects on # of Backups	Reduced dramatically for all schemes

Effects of Increasing MIT on Processing Time	Processing time increases for all schemes except for EVRT where it decreases
Scheme Ordering Based on Processing Time (ordering from worst to best)	1) Fixed-list 2) Circular-list and TQ 3) Partitioning and EVRT 4) LVRT
Lazy Message Cancellation's Effects on Processing Time	No effect

Figure 7.54 Summary of Random Partitioning II with 16 Partitions

7.9.2 16-Partition Case

This experiment was done with three different ways of randomly partitioning the 16-input-node butterfly network into 16 partitions. The partition schemes are shown in the Appendix. Each partition was allocated two processors. The result of one experiment (using the R_{16b} partitioning – see Section **5.5.2**) is shown in Figures **7.52**, and **7.53**; the results of the others are similar. Figure **7.52** shows the number of backups for each of the scheduling schemes as the MIT increases. Figure **7.53** displays the processing times for each of the schemes as the MIT increases.

Once again we have facts that are similar to the horizontal partitioning experi-

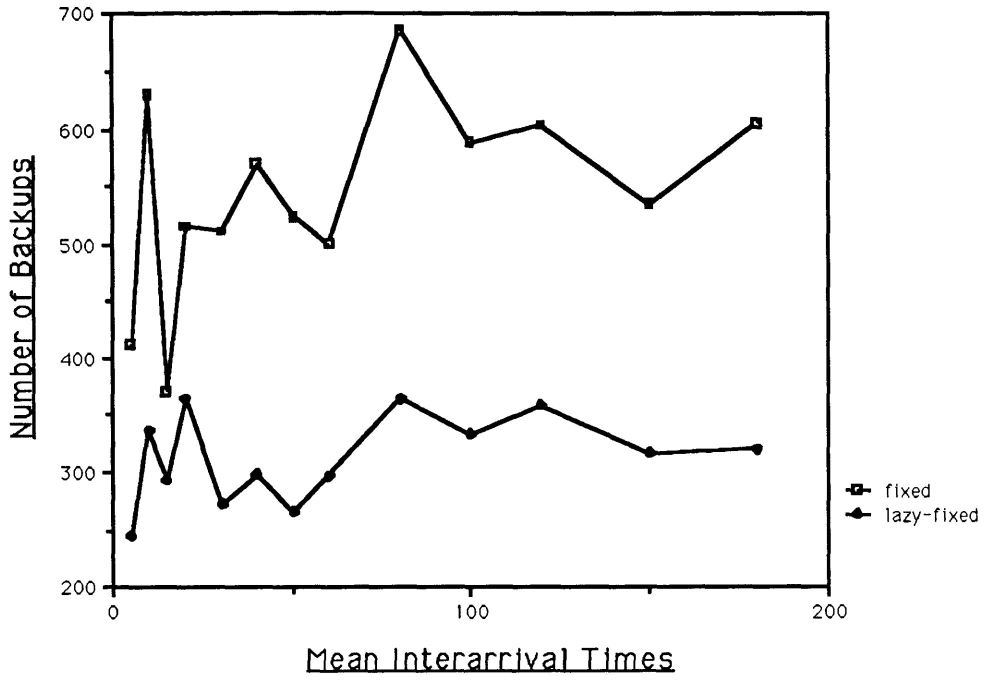


Figure 7.55 Number of Backups versus MIT for R_{16b} using Aggressive or Lazy Cancellation with Fixed-list scheduling

ments. The EVRT scheme performed the best in terms of backups while the LVRT scheme performed the best in terms of processing time. Again, as the MITs increased, the EVRT scheme maintained its performance, while the other schemes that performed worse in terms of backups and processing time. The results of the R_{16b} experiment are summarized in Figure 7.54.

As in the eight-partition case, lazy message cancellation was almost always better than aggressive message cancellation in reducing the number of backups in random partitioning for all of the scheduling schemes. This effect is shown in Figure 7.55.

7.9.3 Discussion

The random partitioning method suffers from approximately the same effects as horizontal partitioning. Summarizing the results:

1. The fixed-list scheme performed poorly
2. The EVRT scheme had the fewest backups

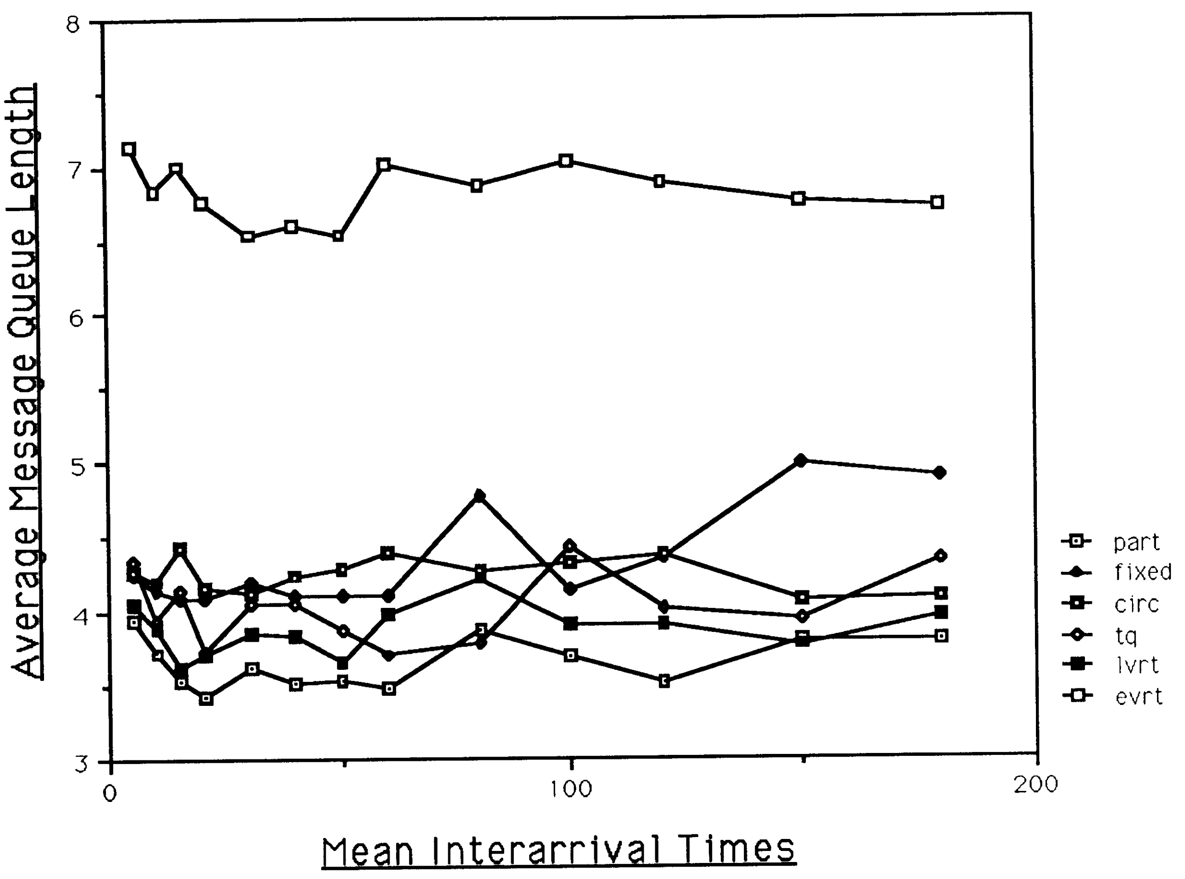


Figure 7.56 Average Message Queue Lengths for R_{8c}

3. The LVRT scheme had the lowest processing times
4. Random partitioning performed poorly
5. Lazy message cancellation improved performance.

In comparing the number of backups and processing times of the random partitioning method to the vertical partitioning, the random partitioning method suffers from more backups and longer processing times. This is due to the aggressive backup effect. EVRT is the best at alleviating this effect as was mentioned in Section 7.5.3 and as confirmed by our results.

Like horizontal partitioning, the random partitioning method also fared poorly with increasing MITs. Again, the EVRT scheme performed the best in terms of fewest backups and maintained or lowered the amount of processing time as the MIT increased.

As in horizontal partitioning, the message queue lengths tended to be much lower than those in experiments from vertical partitioning. This is shown in Figures **7.38**, **7.39**, and **7.56**. Thus the number of backups for the random partitioning method should be greater than for vertical partitioning, where the message queues are longer.

Again, lazy message cancellation also improved performance in all of the scheduling policies because it helped to reduce the aggressive backup effect. This improvement was also shown in Section **7.8** where random partitioning was used on an 8-input-node butterfly network.

Minimum Communication Partitioning with 8 Partitions

Initial Processor Allocation	(8 8 8 8 0 0 0 0) (except static) (4 4 4 4 4 4 4 4) (static)
Effects of Increasing MIT on # of Backups	# of Backups increases in all schemes except for EVRT where it remains constant
Scheme Ordering Based on # of Backups (ordering from worst to best)	1) EVRT 2) Fixed-list, Circular-list, TQ and LVRT 3) Partitioning
Lazy Message Cancellation's Effects on # of Backups	None

Effects of Increasing MIT on Processing Time	Processing Time remained constant
Scheme Ordering Based on Processing Time (ordering from worst to best)	1) EVRT 2) Circular-list 3) Partitioning, Fixed-list, TQ and LVRT
Lazy Message Cancellation's Effects on Processing Time	None

Figure 7.57 Summary of Minimum Communication Partitioning with 8 Partitions

Minumum Communication Partitioning with 12 Partitions

Initial Processor Allocation	(4 4 4 4 4 4 4 4 0 0 0 0) (except static) (3 3 3 3 3 3 3 3 2 2 2 2) (static)
Effects of Increasing MIT on # of Backups	# of Backups increases in all schemes except for EVRT where it remains constant
Scheme Ordering Based on # of Backups (ordering from worst to best)	1) EVRT 2) Fixed-list, Circular-list, TQ and LVRT 3) Partitioning
Lazy Message Cancellation's Effects on # of Backups	None

Effects of Increasing MIT on Processing Time	None
Scheme Ordering Based on Processing Time (ordering from worst to best)	1) Circular-list and EVRT 2) Partitioning 3) Fixed-list, TQ and LVRT
Lazy Message Cancellation's Effects on Processing Time	None

Figure 7.58 Summary of Minimum Communication Partitioning with 12 Partitions

Minumum Communication Partitioning with 24 Partitions

Initial Processor Allocation	(4 4 4 4 4 4 4 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)(except static) (2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)(static)
Effects of Increasing MIT on # of Backups	# of Backups increases in all schemes except for EVRT and static partitioning where it remains constant
Scheme Ordering Based on # of Backups (ordering from worst to best)	1) EVRT 2) LVRT 3) Circular-list 4) Fixed-list and TQ 5) Partitioning
Lazy Message Cancellation's Effects on # of Backups	None

Effects of Increasing MIT on Processing Time	Processing time of Circular-list, TQ, and EVRT decreases
Scheme Ordering Based on Processing Time (ordering from worst to best)	1) Circular-list 2) EVRT 3) TQ 4) Fixed-list and LVRT 5) Partitioning
Lazy Message Cancellation's Effects on Processing Time	None

Figure 7.59 Summary of Minimum Communication Partitioning with 24 Partitions

7.10. Minimum Communications Partitioning

7.10.1 Results

Here we discuss the results of the 16-input-node butterfly network with minimum communications partitioning with MIT between 5 and 180 and with ND equal to 10. The partitioning methods for MC_8, MC_{12}, and MC_{24} are shown in Figures **5.8**, **5.9**, and **5.10**, respectively. The initial processor assignments for the dynamic scheduling policies divided the processors equally between the input partitions. This assignment was chosen because the input partitions had all the work at the start of the simulation; this was an optimal processor assignment.

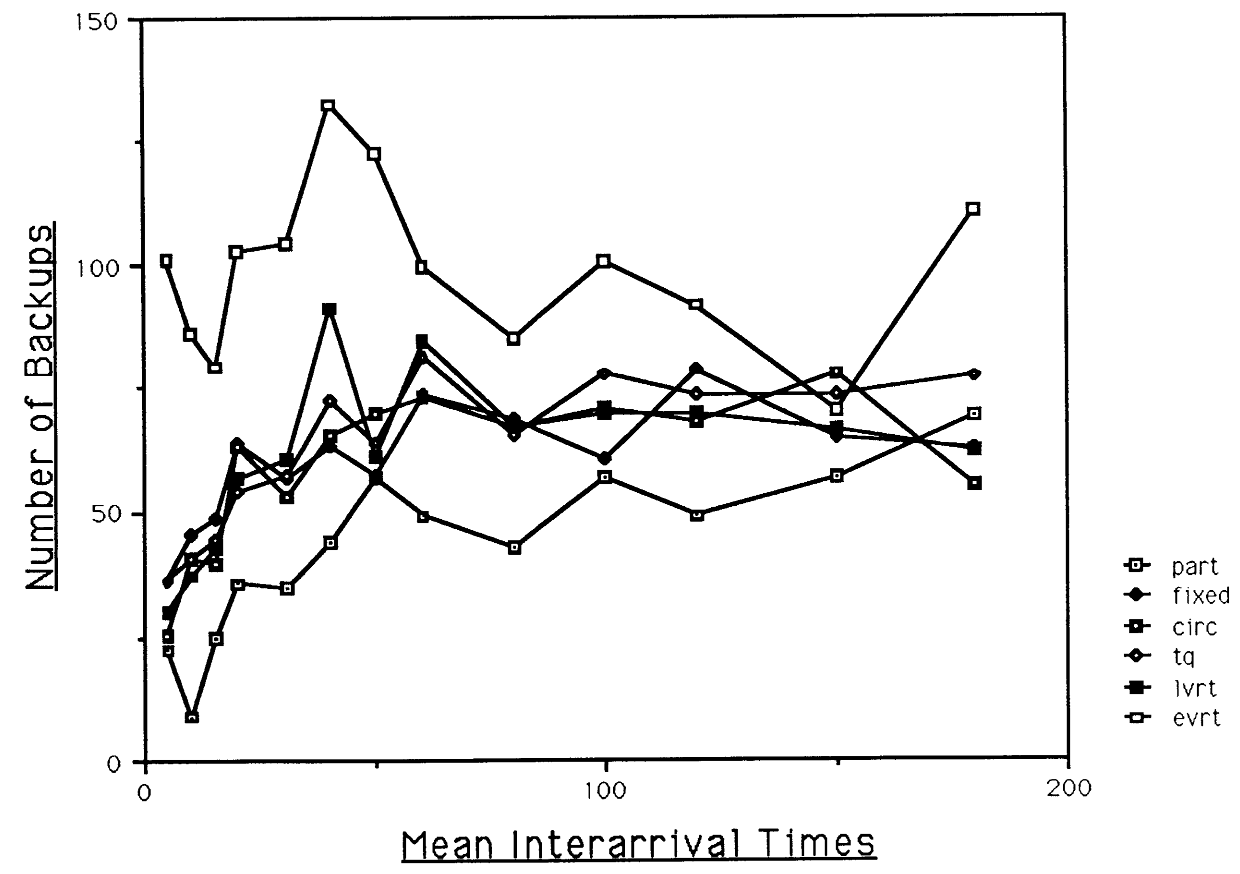


Figure 7.60 Number of Backups versus MIT for MC_8

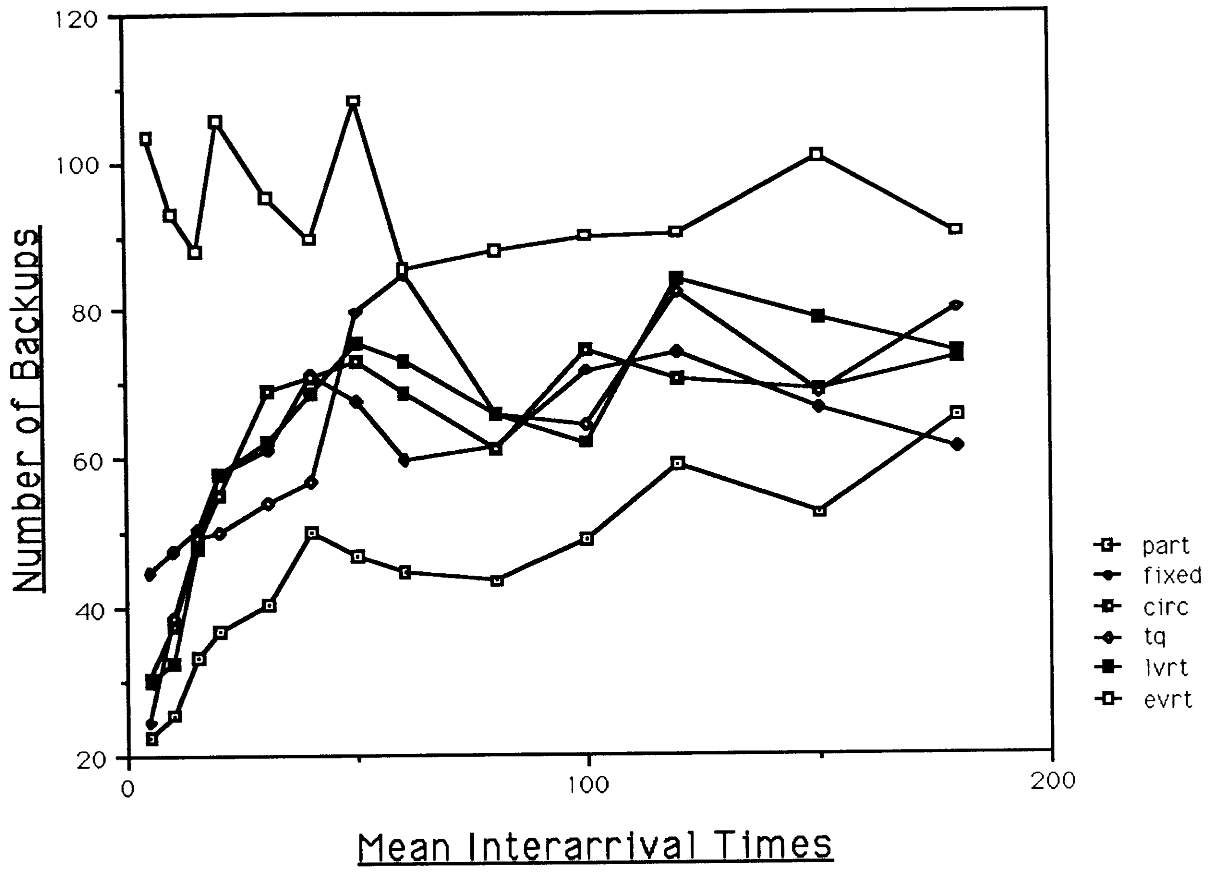


Figure 7.61 Number of Backups versus MIT for MC_{12}

In the static partitioning case for MC_8, each partition started with four processors; this was not optimal, but since each partition had about the same amount of work it was close to optimal given that processors do not move. In the static partitioning case for MC_{12}, the first eight partitions started with three processors and the last four started with two processors; this was not optimal, but since the first eight partitions had to start up and send messages first they were given the extra processors. In the static partitioning case for MC_{24}, the first eight partitions started with two processors and the last sixteen partitions started with one processor; this was not optimal, but since each partition had about the same amount of work it was close to optimal given that processors do not move.

These experiments on MC_8, MC_{12}, and MC_{24} are summarized in Figures 7.57,

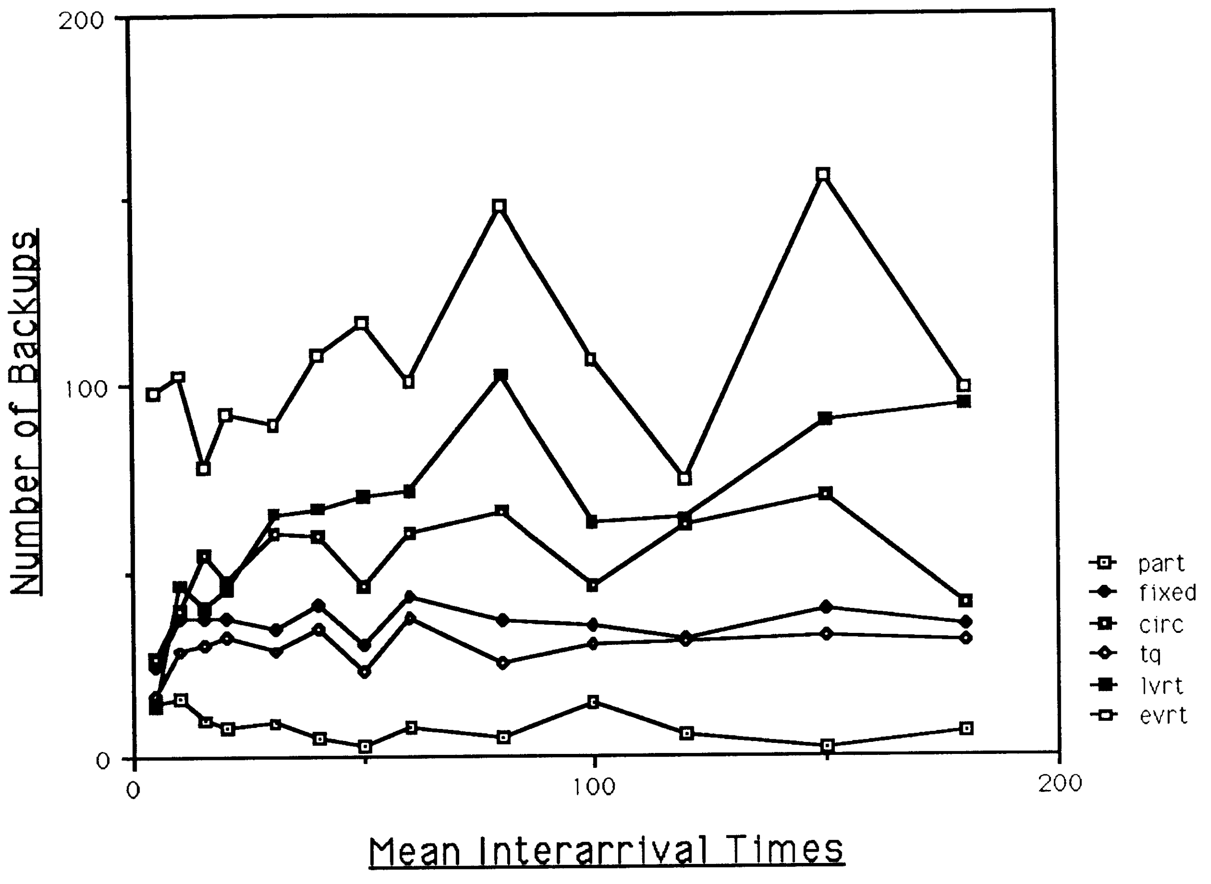


Figure 7.62 Number of Backups versus MIT for MC_{24}

7.58 and **7.59**, respectively. The number of backups for each minimum-communication partitioning method is plotted in Figures **7.60**, **7.61** and **7.62**, respectively. These graphs show the number of backups for each scheduling scheme as the MIT increases. As the MIT increases, the number of backups increases in all of the scheduling schemes except in the static partitioning scheme in MC_{24} and the EVRT scheme (in all the partitioning methods) where the numbers remain constant. The static partitioning scheme has the fewest backups while the EVRT has the most. Lazy message cancellation had no effect on the number of backups, as shown in Figures **7.63**, **7.64** and **7.65**.

Figures **7.66**, **7.67** and **7.68** show the processing times of the different scheduling schemes as the MIT increases. Increasing the MITs does not affect the processing times. In MC_8, the LVRT and the static partitioning schemes had the lowest processing times

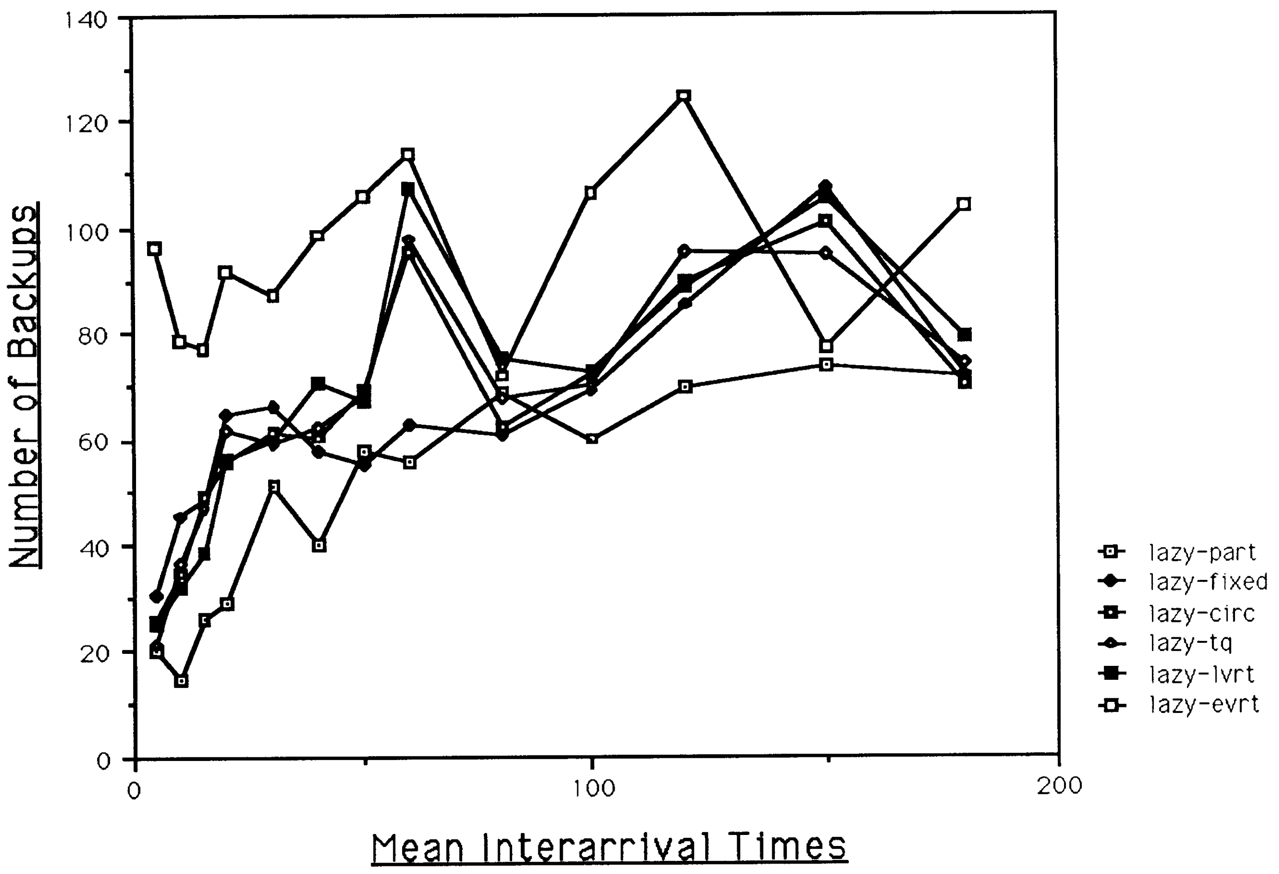


Figure 7.63 Number of Backups versus MIT for Lazy Cancellation for MC_8

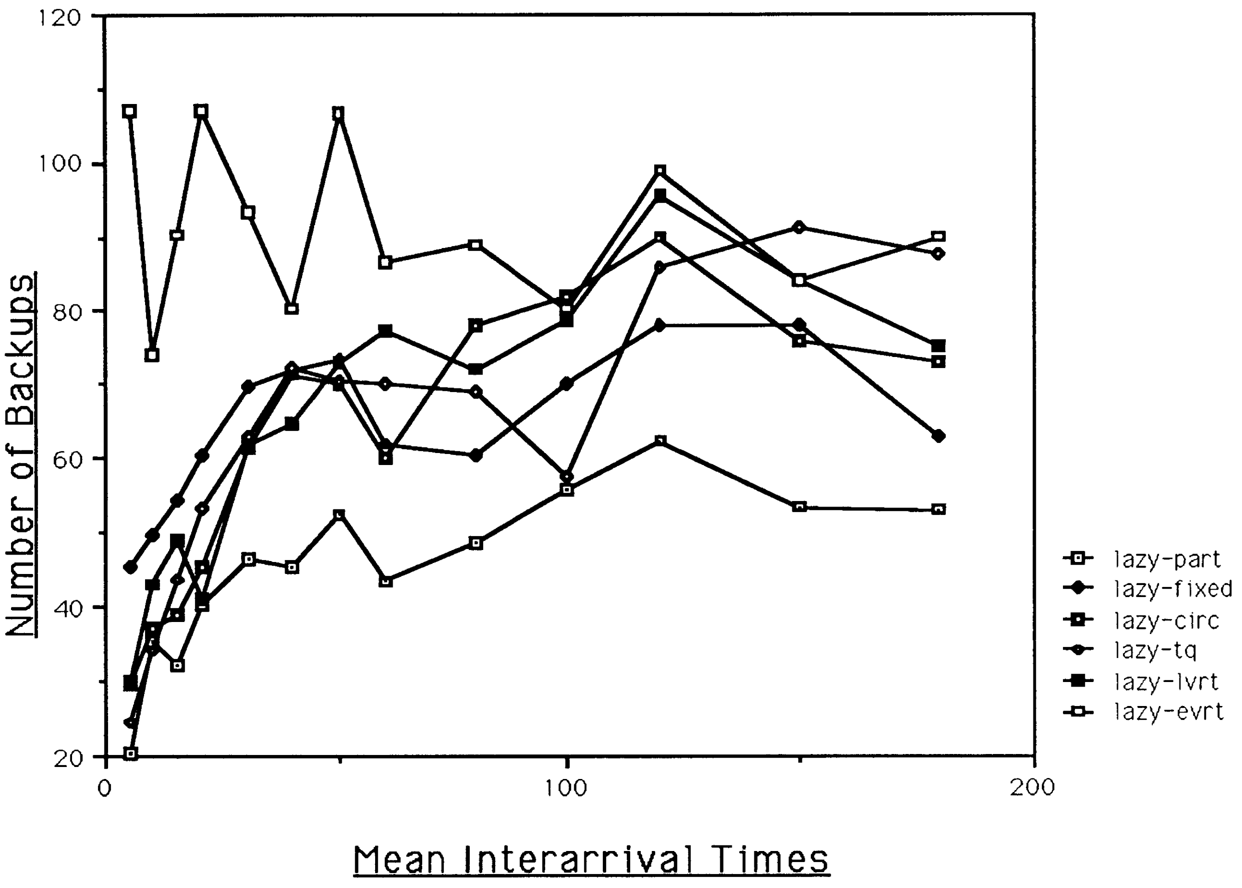


Figure 7.64 Number of Backups versus MIT for Lazy Cancellation for MC_{12}

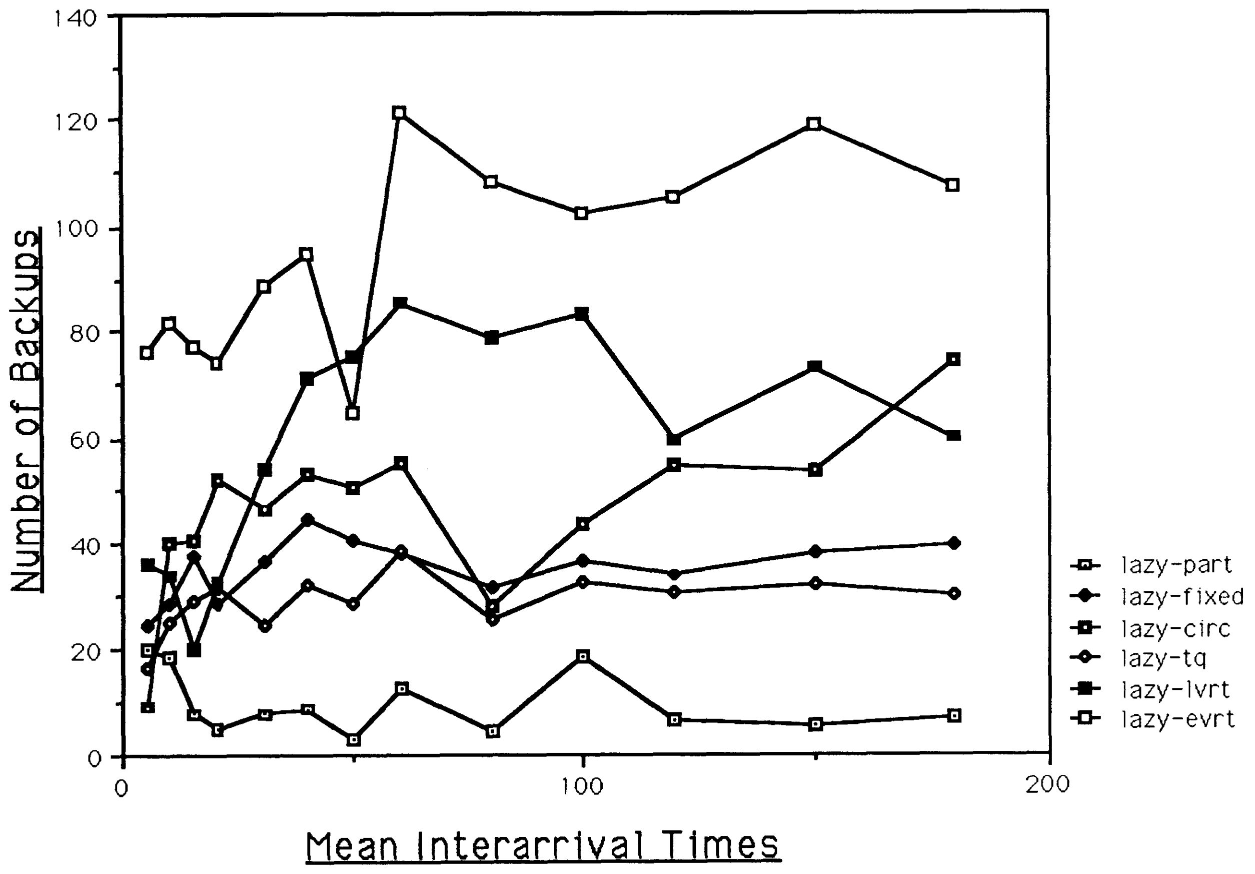


Figure 7.65 Number of Backups versus MIT for Lazy Cancellation for MC_{24}

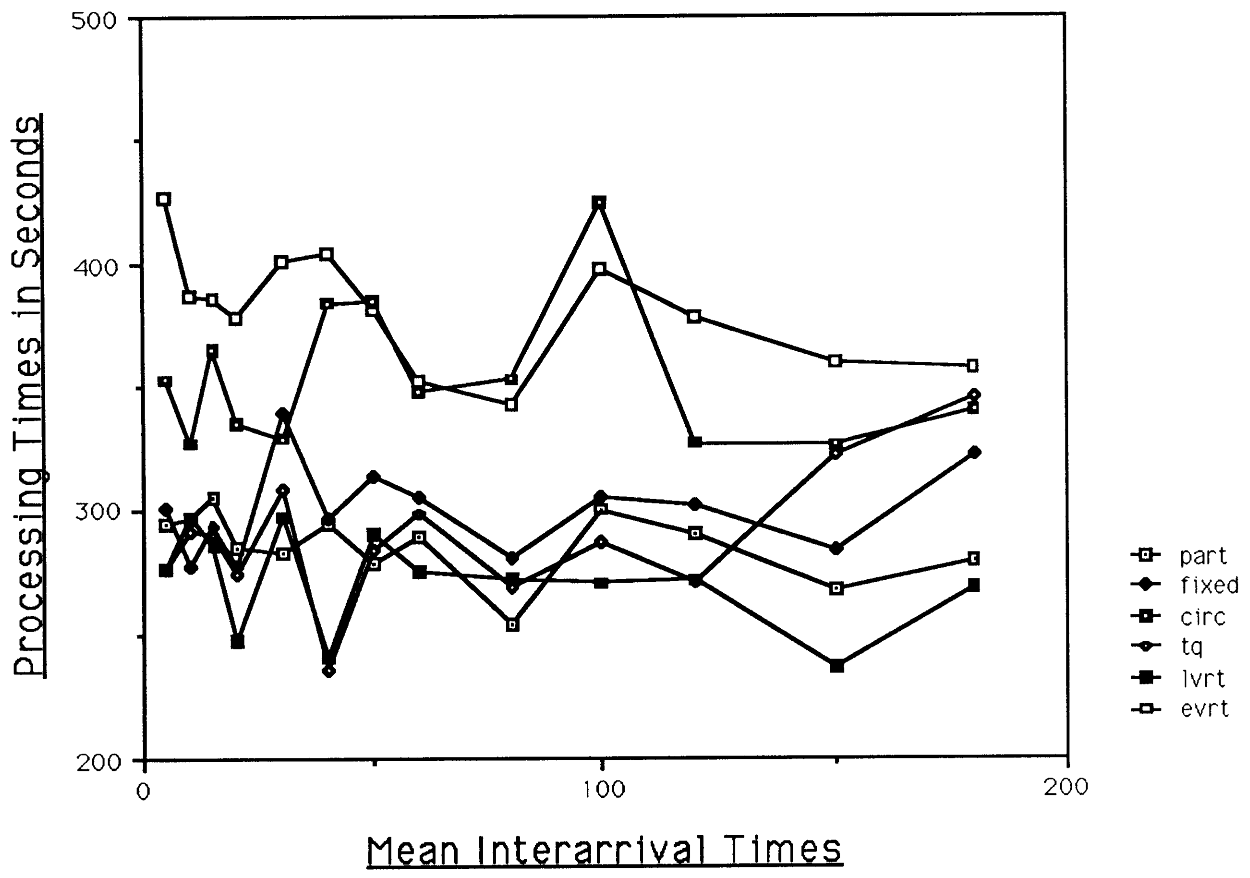


Figure 7.66 Processing Time versus MIT for MC_8

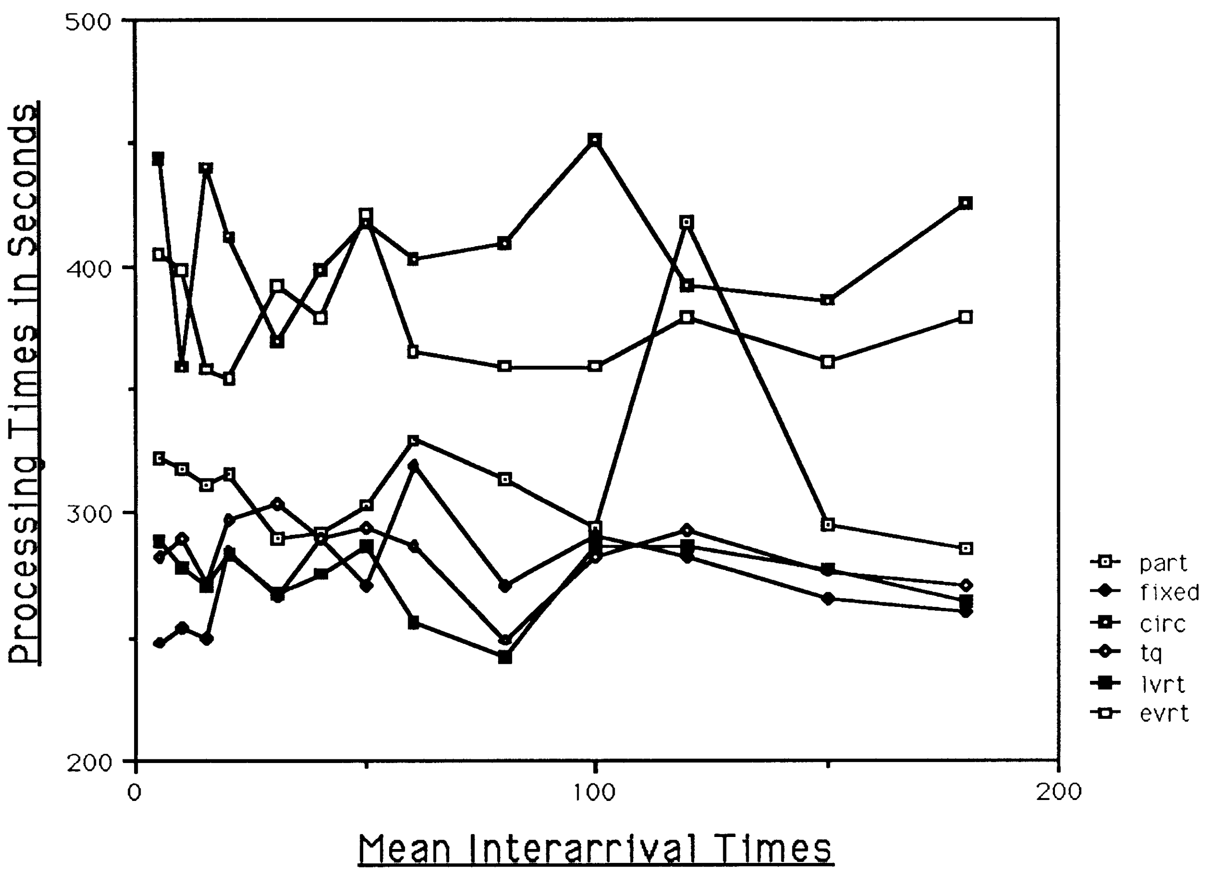


Figure 7.67 Processing Time versus MIT for MC_{12}

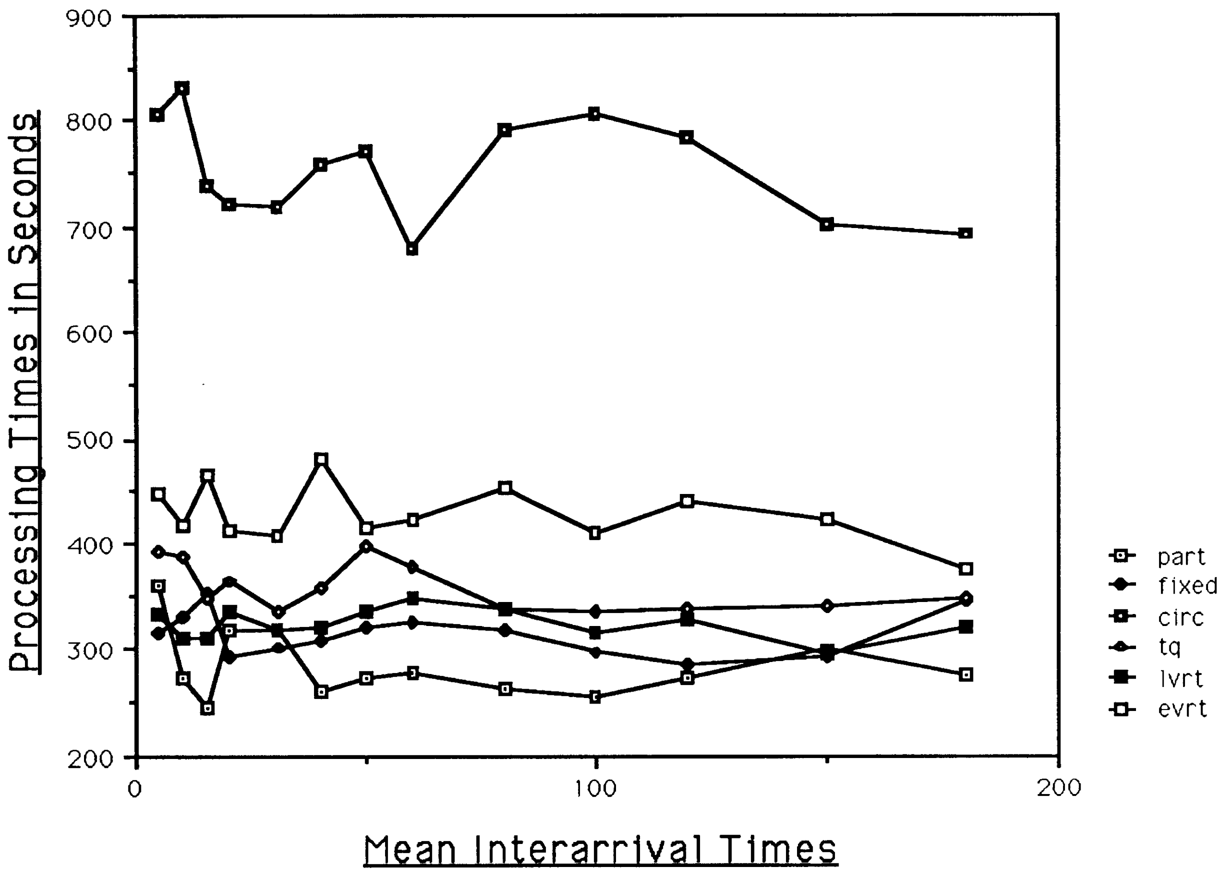


Figure 7.68 Processing Time versus MIT for MC_{24}

while the EVRT scheme had the highest. In MC_{12}, the LVRT and fixed-list schemes had the lowest processing times while the EVRT and circular-list schemes had the highest. In MC_{24}, the static partitioning scheme had the lowest processing times while the circular-list scheme had the highest. In all cases, lazy message cancellation did not change the performance of any scheme.

7.10.2 Partition Saturation

Figures **7.60**, **7.61** and **7.62** show that static partitioning has fewer backups than all of the dynamic scheduling schemes. In this section we provide an explanation for this phenomenon. It is caused by an overabundance of processors in a partition called partition saturation which in turn causes additional backups. This does not usually occur in static partitioning since partitions are given a small constant number of processors. On the other hand, in dynamic scheduling schemes, clustering of many processors in a single partition is common.

7.10.2.1 Theory

When a partition has too many processors, backup will occur due to partition saturation. This phenomenon happens regularly when we use dynamic repartitioning strategies, i.e. the fixed-list, circular-list, longest task-queue, lowest LVRT, or lowest EVRT strategies. Let us return to the model of Section **7.2** along with its assumptions, except let us assume that the four tasks t_1, t_2, u_1, and u_2 in our model are in one partition. If we assign only one processor to the partition, then no backup will occur; however, as we increase processors in the partition, more backups will occur.

Let us take a look at the model when three processors, A, B, and C, are assigned to the partition and ND=0. Let the input messages to t_1 be $((5,u_2)(10,u_2))$ and the messages into u_1 be $((1,u_2)(2,u_2))$, as shown in Figure **7.69**. At first one processor is assigned to each of t_1 and u_1 (let us say that processor A is assigned to t_1 and processor B is assigned to u_1), while the third processor remains suspended because there is no work to do. Once one of processors A or B finishes executing a task, there will be

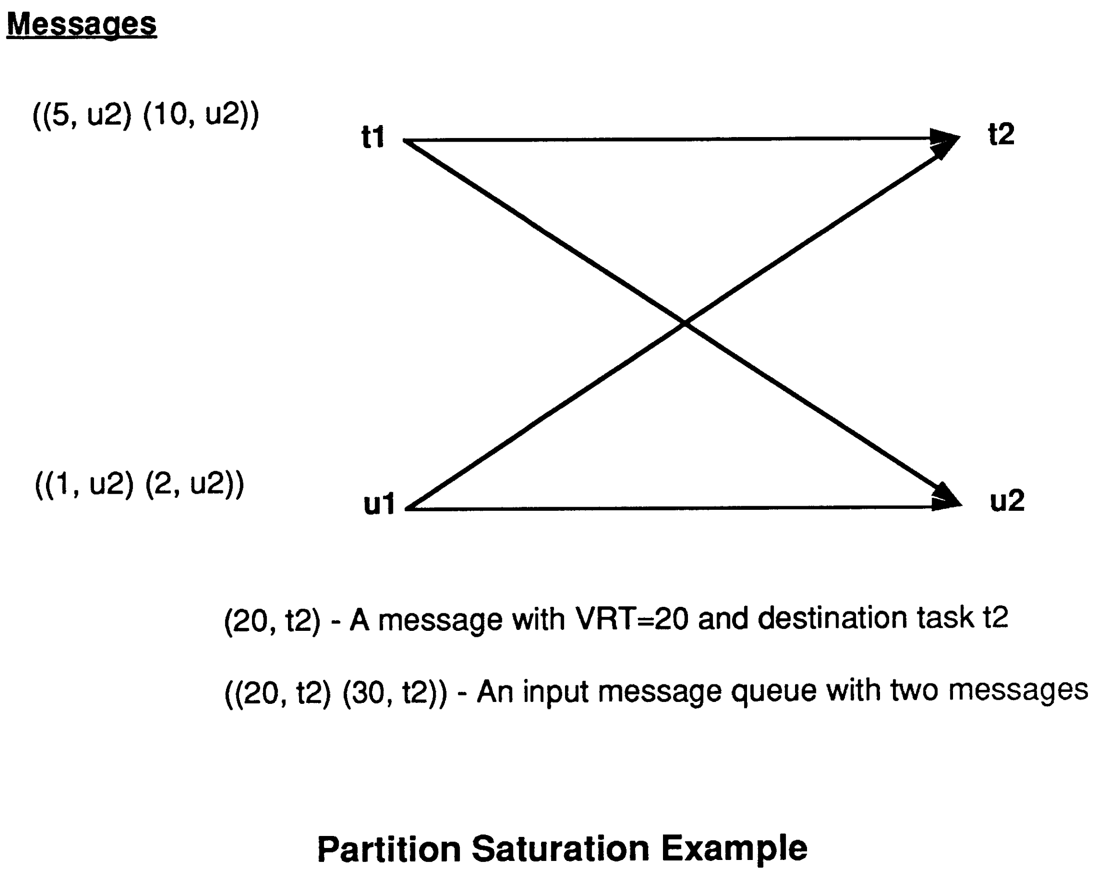

Messages
((5, u2) (10, u2))
t1
t2
((1, u2) (2, u2))
u1
u2
(20, t2) - A message with VRT=20 and destination task t2
((20, t2) (30, t2)) - An input message queue with two messages
Partition Saturation Example

Figure 7.69 Example of the Partition Saturation Effect

a message at task u_2 and the third processor, C, will become active. If processor B finishes first, then there will be no problem since processor C will process (1,u_2) at u_2 while processor B processes (2,u_2) at u_1. On the other hand, if processor A finishes first, then a backup will occur because processor A will process (10,u_2) at t_1 while processor C will process (5,u_2) at u_2. Once processor B finishes processing the message (1,u_2) at u_1, task u_2 will be forced to back up.

This problem only becomes worse as we increase the number of processors to four. In this case, there is one processor for each task. Hence, there is no attention paid to the VRTs of messages when scheduling tasks. Additional backups can occur even with only two processors in the partition if the nodal delay is small in relation to the MIT. In our example with ND=0, assigning two processors to the partition can cause backups, while if the ND=6, then no backup will ever occur with two processors. This is another example of virtual-time delay effects (discussed in Section **7.5**); however, in this case the virtual-time delays aggravate the partition saturation effect. Therefore partition saturation – assigning too many processors to one partition – can lead to additional backups.

7.10.2.2 Experiments

The partition saturation effect was studied on the minimum communications partitioning method by varying the number of processors in one partition. Since partition saturation only occurred when there was an abundance of processors in a partition, it was necessary to determine if many processors did cluster at a single partition. A series of experiments tested whether many processors did cluster at a single partition by tracing the number of processors per partition at different points in time.

This section describes experiments that confirm our theories of partition saturation. When a partition has a large ratio of processors to tasks, the rate of backups will increase due to partition saturation. The number of backups grows rapidly as the number of processors approaches the number of tasks in the partition. We are interested in studying this phenomenon because in experiments involving minimum communications partitioning, the number of backups for dynamic scheduling schemes was at least 10% more than that of the static scheduling schemes. We attribute this phenomenon to partition saturation.

In order to show that this is true we need to show:

1. As the number of processors approaches the number of tasks in a partition, the number of backups grows quickly.
2. In the dynamic scheduling schemes, processors are not evenly distributed amongst partitions. This is due to processor movement between partitions.

We performed two experiments to test our theories. We used the MC_8 partitioning method illustrated in Figure **5.8**. Each partition had twelve tasks. We wanted to test the effects on one partition as we increased the number of processors from one to twelve.

Since we did not want the number of processors in the test partition to change, we used the static partitioning scheme. We allocated four processors to each of the four input partitions and one processor to three of the output partitions. In this manner the four input partitions ran exactly as they would in the static partitioning scheme with four processors in each partition.

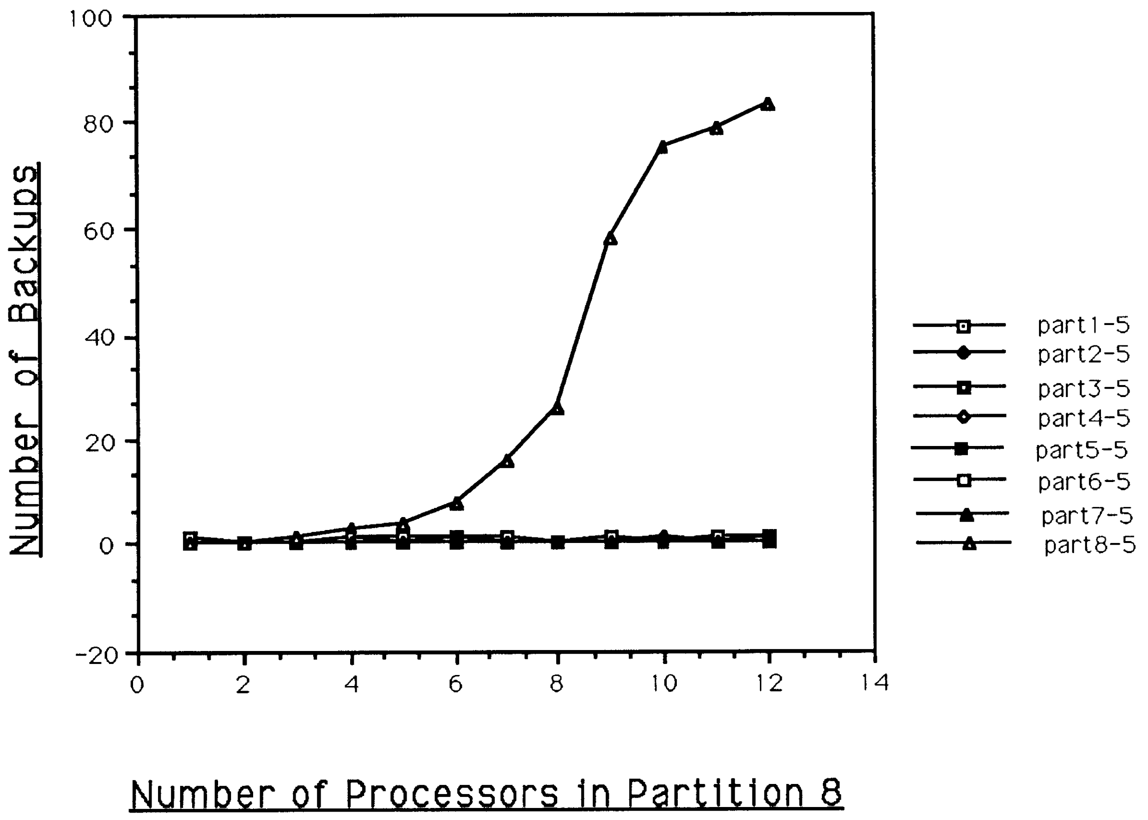


Figure 7.70 Number of Backups in each Partition while varying processors from 1 to 12 and MIT=5

In one of the output partitions we varied the number of processors from one to twelve. The number of backups for each partition as we vary the processors in partition eight are displayed in Figure **7.70**. This experiment was done with MIT=5; in the diagram, the caption "part$\underline{N}$-5" represents partition $\underline{N}$'s backups with MIT=5 as

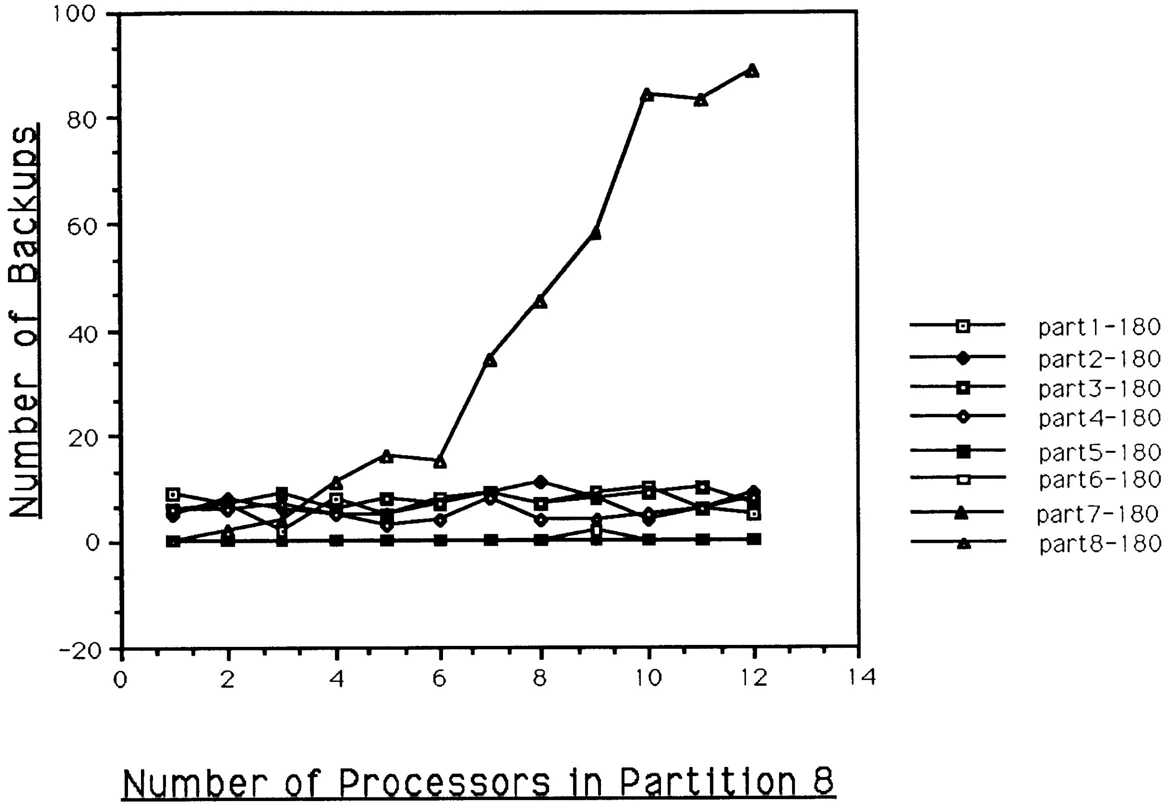


Figure 7.71 Number of Backups in each Partition while varying processors from 1 to 12 and MIT=180

we increase processors from 1 to 12. The figure shows that the number of backups increases rapidly when the number of processors in the partition is between 5 and 10. For the range of 11-12 processors the number of backups levels off. Illustrating that this attribute is not MIT-dependent, Figure **7.71** shows the same experiment with MIT=180. Notice there is little change in the results.

The second experiment was used to determine the clustering of processors in partitions for the dynamic scheduling schemes. In other words we wanted to show that it was uncommon for processors to be approximately evenly distributed among partitions. To show clustering of processors, we performed a trace of the number of processors in

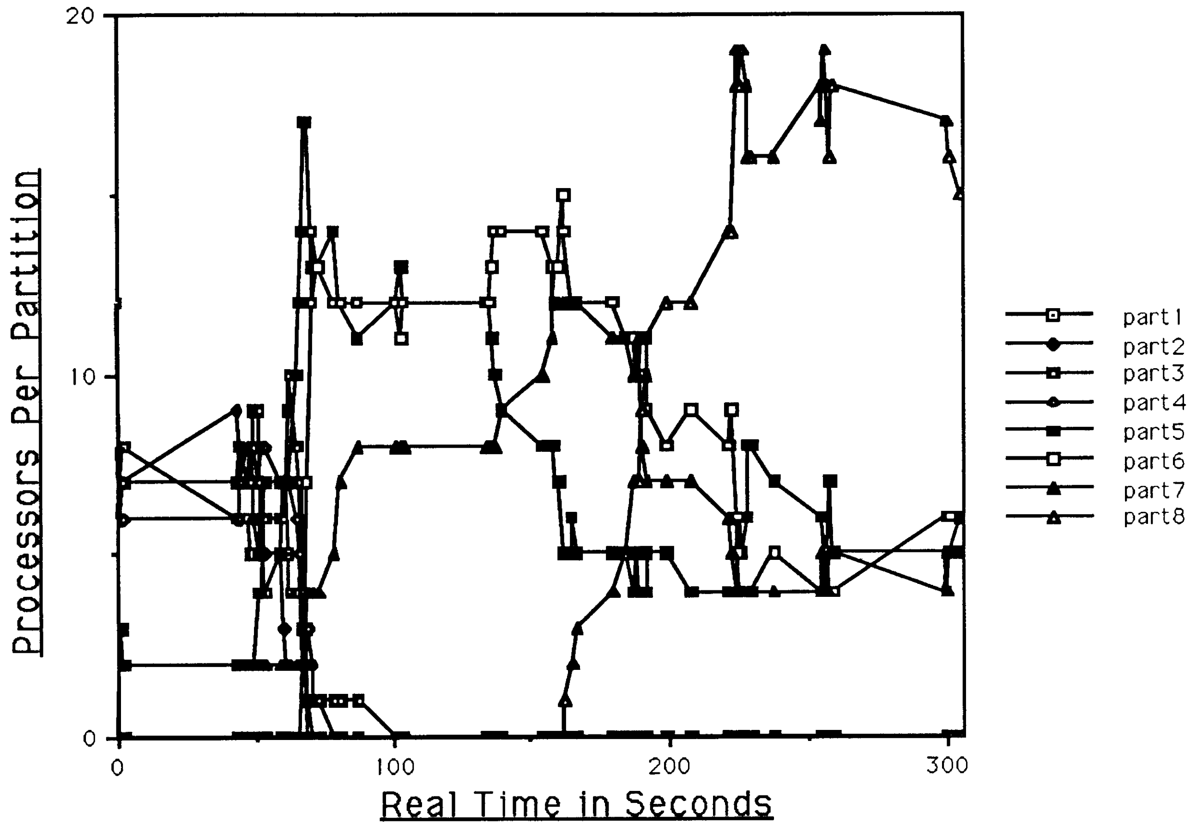


Figure 7.72 Number of Processors in each Partition in MC_8 using the Fixed-list scheme, with ND=10 and MIT=5

each partition at each instant of time, which is shown in Figure 7.72 for dynamic scheduling schemes only. This processor trace showed that the number of processors in each partition is usually nowhere near the average of four. This clustering would clearly trigger the partition saturation effect shown in Figures 7.70 and 7.71, leading to an increase in backups for dynamic scheduling schemes.

We now explain how partition saturation occurs in the dynamic schemes. Let us examine how partition saturation affects the MC_8 case. Partition saturation occurs whenever there is an overabundance of processors in a partition. When using dynamic scheduling schemes, partition saturation occurs at the start of the simulation in the four input partitions (those containing the input nodes) and at the end of the simulation in the four output partitions (those containing the output nodes).

When the simulation starts up in the MC_8 case, there are eight processors assigned to the four input partitions. On the other hand, each partition contains only eight active tasks (four are drivers); therefore, the eight processors would be an overabundance of processors. This is a clear case of partition saturation. Even if all eight processors were not initially assigned to the four input partitions, they would immediately migrate there because those are the only partitions that have work.

When the simulation is in its later stages, all 32 processors will reside in the four output partitions, because the input partitions have finished all their work by then. This is an average of eight processors in each partition. However, each output partition contains only 12 tasks which means once again there is an overabundance of processors.

Excluding the early and later stages of the simulation, there are many times when partitions contain eight or more processors. This actually occurs in all dynamic scheduling schemes and is shown in **7.72**. This bunching up of the processors causes more backups due to the partition saturation effect. Given the initial processor assignments that were used for the different partitioning methods, dynamic scheduling schemes will have more backups than static scheduling schemes due to their higher susceptibility to partition saturation.

Partition saturation affects MC_{12} and MC_{24} in the same way as MC_8, since both partitioning methods have input and output partitions that have tasks in multiple precedence classes. One difference is that in MC_{12} and MC_{24} the partitions generally contain fewer tasks; therefore, partition saturation will occur at a partition with fewer processors.

The static scheduling scheme was executed with an initial processor allocation of 4 or fewer processors per partition. Since processors do not relocate in the static partitioning scheme, static partitioning always operated with 4 or fewer processors in Figures **7.70** and **7.71**. This meant that very few backups would occur. This explains why static partitioning generally performed 10% better than the dynamic scheduling schemes.

7.10.3 Discussion

The experimental results showed the following effects on the minimum communications partitioning method:

1. The Synchronization Effect
2. The Partition Saturation Effect
3. The Virtual-Time Delay Effect
4. The Message Queue Length Effect

Summarizing the results:

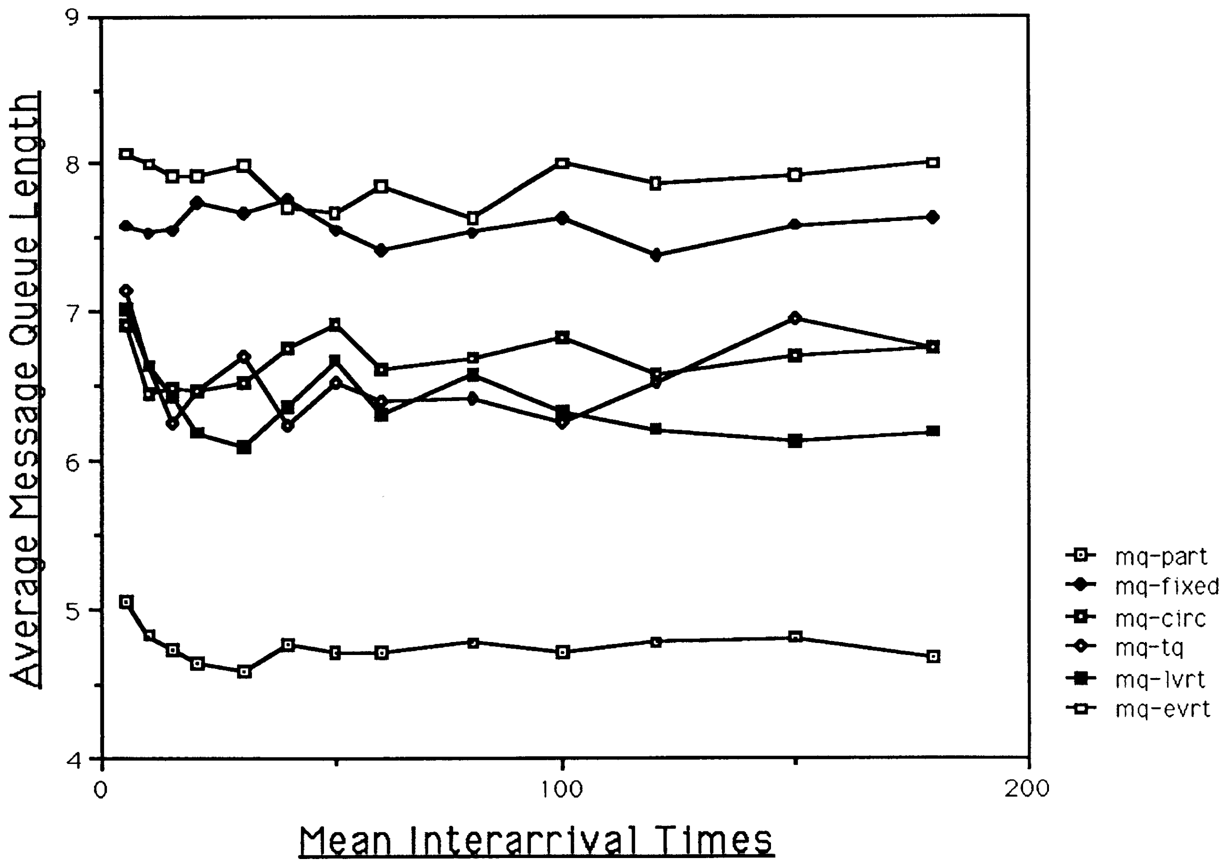


Figure 7.73 Average Message Queue Lengths for MC_8

1. The EVRT scheme had the most backups while the static partitioning scheme had the fewest
2. The EVRT scheme had high, if not the highest processing times, while static partitioning had low, if not the lowest processing times
3. Minimum communication partitioning clearly outperformed horizontal and random partitioning in the number of backups (illustrated in Figures **7.60**, **7.61**, **7.62**, **7.7**, **7.9**, **7.48**, and **7.52**) and in processing times (illustrated in Figures **7.66**, **7.67**, **7.68**, **7.8**, **7.10**, **7.49**, and **7.53**). It performed about the same as vertical partitioning in the number of backups (illustrated in Figure **7.12**) and in processing times (illustrated in Figure **7.13**).
4. Increasing virtual-time delays degraded performance, except for the EVRT scheme and the static partitioning scheme for MC_{24}.
5. Message queue lengths were much higher than those of horizontal or random partitioning experiments (illustrated in Figures **7.38**, **7.56**, **7.73**), but were about the same as those of vertical partitioning experiments (illustrated in Figures **7.39**).
6. Lazy message cancellation had no effect.

The six previous statements proved to be true of all of the Minimum Communications partitioning methods. Each partitioning method suffered from partition saturation which was aggravated by increases in virtual-time delays (excluding the exceptions indicated in item 4). One exception was that the static partitioning scheme for MC_{24} did not seem susceptible to virtual-time delays. This was probably because most of the partitions had only one processor. Partition saturation does not occur when there is only one processor in a partition; therefore, the number of backups was not affected by virtual-time delays.

Note that the trend of each scheduling scheme is clearer as the number of partitions increases. In other words, the trend is clearer in MC_{12} than in MC_8, while the trend is much clearer in MC_{24} than either MC_{12} or MC_8. This also was the case in horizontal partitioning as seen in Section **7.5.2**.

Minimum communication partitioning's good performance, as well as static partitioning's outperformance of the optimized dynamic strategies, lead us to believe that minimum communication partitioning is one of the better forms of partitioning for the network simulation.

The EVRT scheme did not perform well because it does not benefit from the synchronization effect. Since the EVRT scheme does not synchronize the tasks within the partition in terms of LVRT, more backups result.

As in vertical partitioning, we also attribute the good performance of minimum communication partitioning to the lack of an aggressive backup effect. Lazy message cancellation had no effect since the aggressive backup effect was not present.

The message queue lengths in the minimum communications partitioning experiments were much higher than those of the horizontal or random partitioning experiments, but were about the same as vertical partitioning experiments. This is shown in Figure **7.73**. This correlated to fewer backups, illustrating the relationship of backups to message queue length.

As shown in the example of Figure **7.69**, the partition saturation effect is compounded by virtual-time delays (low ND in comparison to MIT). In this example backups due to partition saturation occurred when ND=0, but were absent when ND=6. Therefore minimum-communications partitioning is susceptible to virtual-time delays. This last fact is confirmed by examining Figure **7.71** where the input partitions (1-4) had slightly more backups in the MIT=180 case than in the MIT=5 case.

Susceptibility to virtual-time delays explains why all the scheduling schemes showed a slight increase in backups due to increasing MIT. The minimum communications partitioning method fared poorly due to virtual time delays (increasing MITs). As the MITs increased from 5 to 180 virtual-time units, each scheduling scheme except EVRT had more backups. This is illustrated in Figures **7.60**, **7.61**, and **7.62**. Once again the EVRT scheme was not affected by increasing MITs.

In our experiments involving minimum-communication partitioning, the dynamic scheduling schemes had at least 10% more backups than the static scheduling schemes. We attribute this phenomenon to partition saturation which rarely occurs in static partitioning, but does occur in dynamic scheduling.

In our experiments, partition saturation rarely occurs in static partitioning because each partition always has less than one third as many processors as tasks. This is confirmed by Figure **7.70** which shows that in MC_8 with any scheduling scheme, when the number of processors is less than five, partition saturation is rare. When the number of processors is less than five, partition saturation only occurs when ND is small relative to MIT (compare **7.70** and **7.71**). We expect similar results for MC_{12} and MC_{24} when the number of processors is less than one third the number of tasks.

However, partition saturation occurs in dynamic scheduling when there are many processors in each partition. This is confirmed by Figure **7.70** which shows that backups become more likely in MC_8 when the number of processors exceeds five. Partitions can often have more than five processors when dynamic scheduling is used. This is shown in Figure **7.72**.

In minimum communications partitioning the synchronization effect in conjunction with non-zero nodal delay helps the simulation performance. Recall that the synchronization effect maintains similar task LVTs within a partition. Therefore, when nodal delay is non-zero, processors prefer to process nodes in the lower precedence classes first. This results in fewer backups.

Additionally the aggressive backup effect, where a backup causes additional backups, does not occur in minimum communications partitioning. The aggressive backup effect cannot occur in minimum communications partitioning because no task receives messages from both inside and outside its partition.

7.11. Continuous Dynamic Repartitioning

In this section we analyze the continuous dynamic repartitioning scheme. The continuous scheme appears to be very similar to the non-partitioning case, since processors are free to move to different partitions after each task execution. We present the following three experiments for the continuous scheme that determine:

1. The relationship of backups to different partitioning methods.
2. The relationship of backups to the virtual time delays.
3. The relationship of backups to the number of processors.

7.11.1 Experiments

The continuous dynamic repartitioning effects were studied on each of the partitioning methods. A series of experiments were run on the vertical partitioning method to help determine why the continuous dynamic repartitioning scheme fared so poorly compared to other scheduling schemes in vertical partitioning. We are interested in studying this scheduling scheme because it is similar to the non-partitioning case and because its performance degenerates rapidly as the MITs increase. We want to determine:

1. Whether the continuous scheme essentially ignores partition boundaries since the processors in the scheduling scheme search for the partition with the lowest LVRT and relocate there after each task execution. To determine this, we ran continuous dynamic repartitioning on each of the partitioning methods while increasing the MIT with ND=10.
2. The effect of the MIT:ND ratio on the performance of the continuous dynamic scheduling scheme. To determine this, we varied the ND in vertical partitioning using continuous dynamic scheduling. Normally the ND is 10; we also tried 1000, 100 and 0 while varying the MIT. This differs from the experiment referred to in (1) because (1) examined the effect of varying MIT while fixing ND=10 for all the different partitioning methods.
3. How the number of backups changes as the number of processors increases. To determine this, we ran continuous dynamic repartitioning on the vertical partitioning and varied the number of processors in the system between 8, 16, and 32 with ND=10.

The results of the first experiment for the aggressive and lazy message cancellation

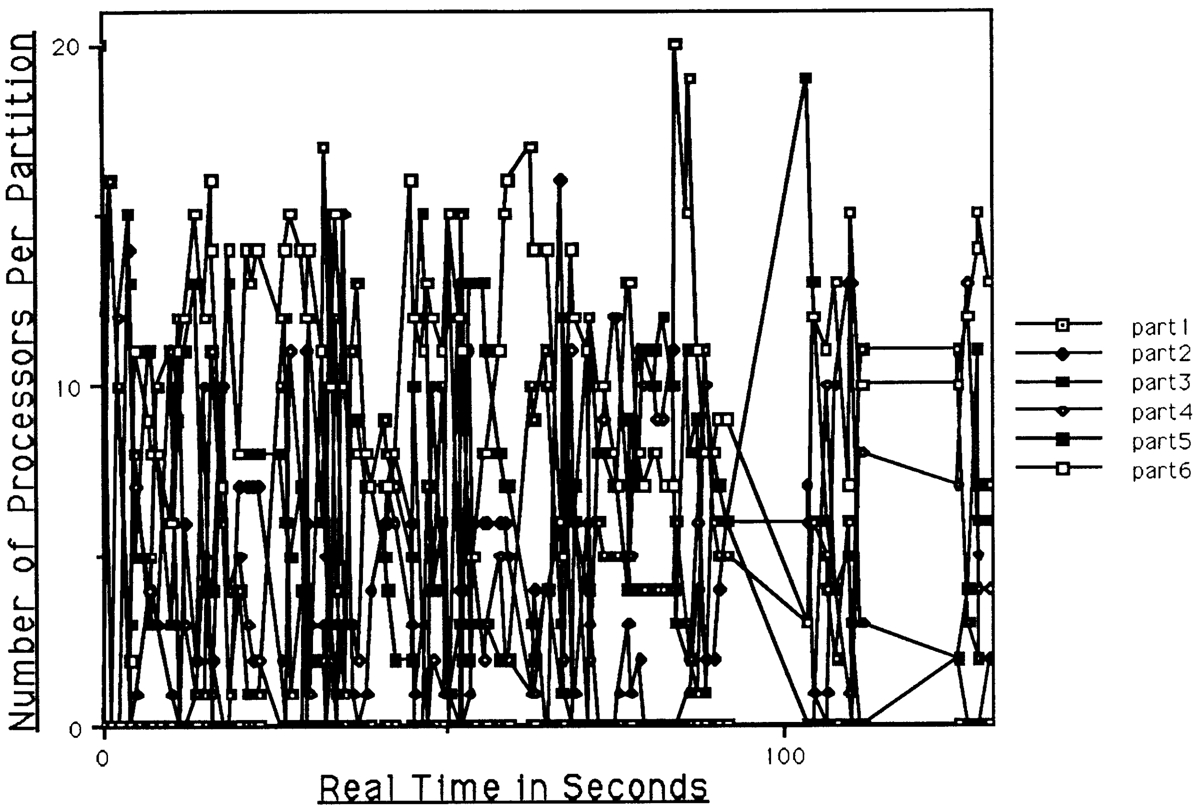


Figure 7.74 Number of Processors in each Partition in V_6 using the Continuous scheme, with ND=0 and MIT=15000

cases are shown in Figures **7.75** and **7.76** respectively. We notice that the performance of the continuous dynamic repartitioning scheme is affected by the partitioning method used.

An additional question that the first experiment raises is why the continuous scheme performs better in vertical partitioning than any other partitioning method. This is caused by the problem in the continuous scheme presented in Section **6.5**. Recall that a "race condition" could arise in the continuous scheme when two processors simultaneously look for the partition with the lowest LVRT in order to relocate there. Since the group list is not locked, both processors could decide to relocate to the same partition, leading to an uneven clustering of processors in partitions. This clustering

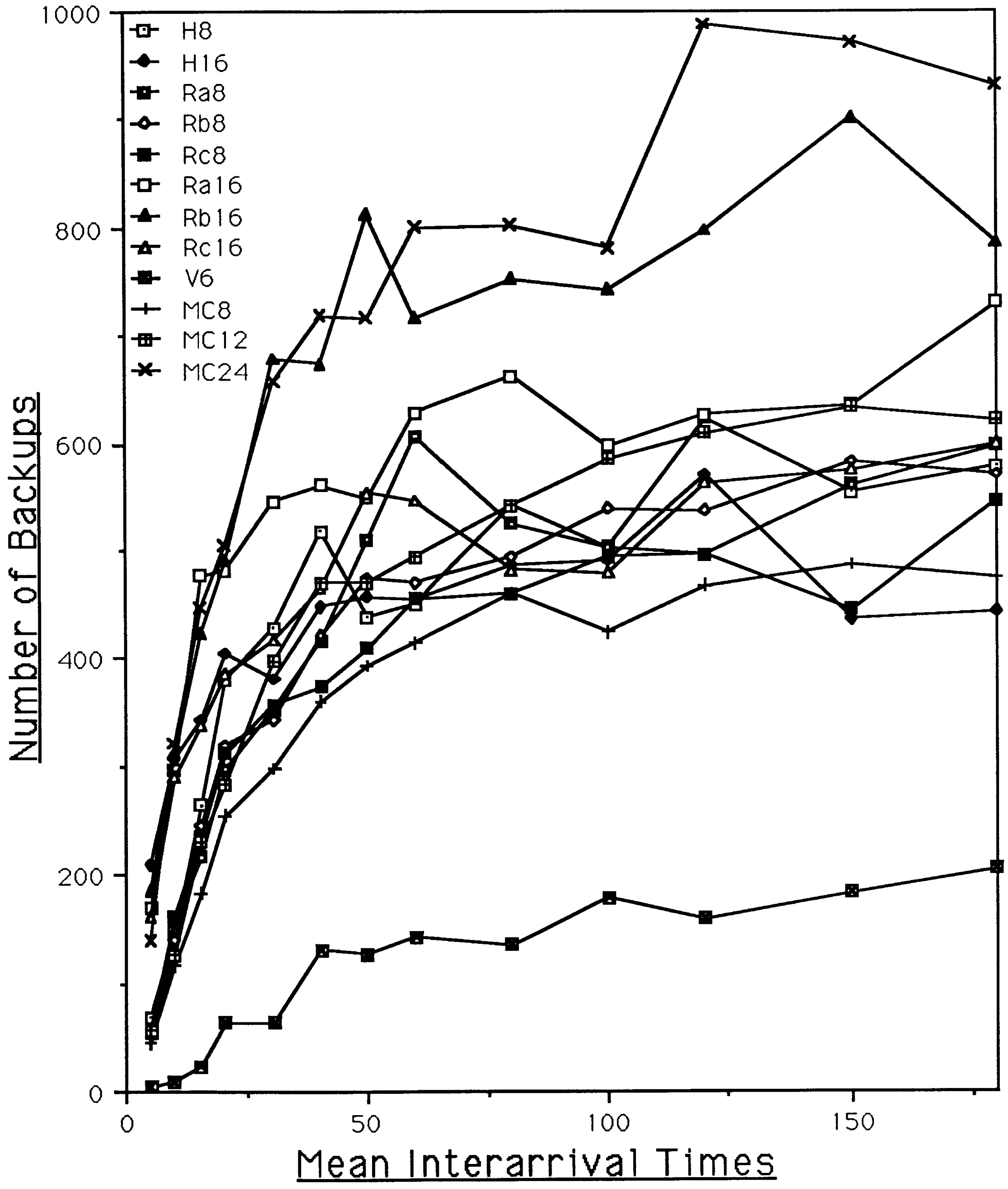


Figure 7.75 Number of Backups for Continuous Dynamic Repartitioning for all Partitioning Methods

effect is shown for the continuous scheme (using vertical partitioning) in Figure **7.74**.

This uneven clustering can increase backups, as discussed in Section **7.10.4**. This effect is especially pronounced when a partition includes more than one precedence class. Let us consider an example where two processors A and B are relocated to the same partition. This partition has two tasks T_1 and T_2, where T_1 is in the higher precedence class and has a link to T_2. There are two cases:

1. T_1's LVRT $<$ T_2's LVRT or
2. T_1's LVRT $>$ T_2's LVRT.

In the first case processor A will be assigned to T_1 and processor B will be assigned to T_2. Let us assume that the ND is smaller than the difference in the two tasks' LVRTs. If processor B completes processing task T_2 before processor A completes processing T_1 and as a result of processing T_1, T_1 sends a message to T_2, then backup will occur. The second case is safe. On the other hand in vertical partitioning all of the tasks are in the same precedence class and no links connect them; therefore, this problem will never occur. Thus the number of backups in the vertical partitioning case will be less than any of the other partitioning strategies.

Another relevant factor appears to be the partitioning size. For instance MC_8 and even the R_8's look much better than MC_{24}. Therefore, one could speculate that larger partitions can absorb a few extra processors with less of a partition saturation effect than smaller partitions.

Since continuous dynamic repartitioning performed the best with vertical partitioning, all subsequent experiments were run only on vertical partitioning.

Figures **7.75** and **7.76** show the effects of virtual-time delay (through variations on MIT) on the continuous dynamic repartitioning scheme using all the different partitioning methods with ND=10. Figure **7.77** shows the number of backups for the continuous scheme with ND=0, 10, 100, and 1000 on V_6. These experiments were used to show the effects of virtual-time delays on the continuous scheme through variations of ND. There are two extreme cases ND=0 and ND=1000. In these cases the ratio of

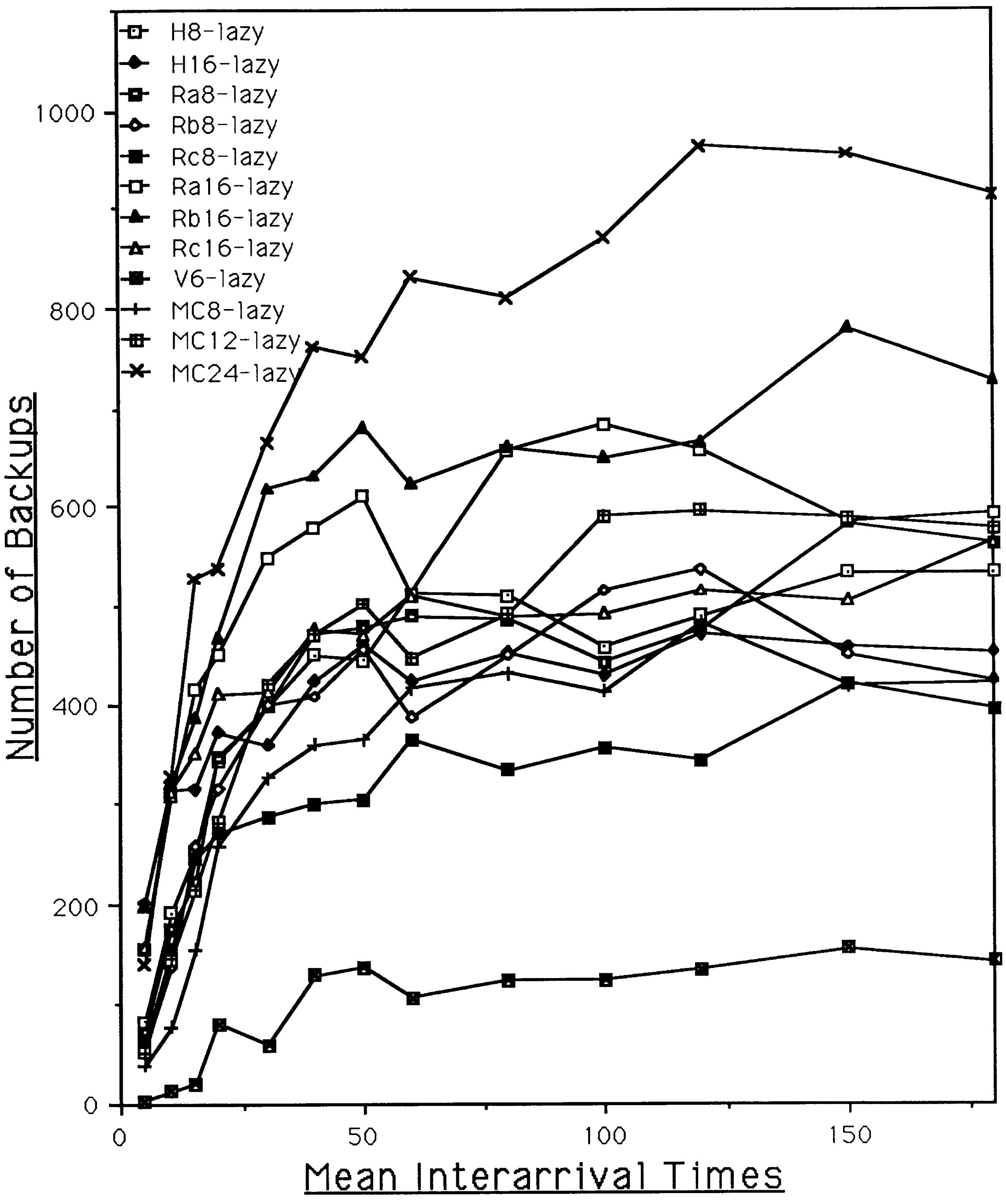


Figure 7.76 Number of Backups for Continuous Dynamic Repartitioning for all Partitioning Methods using Lazy Cancellation

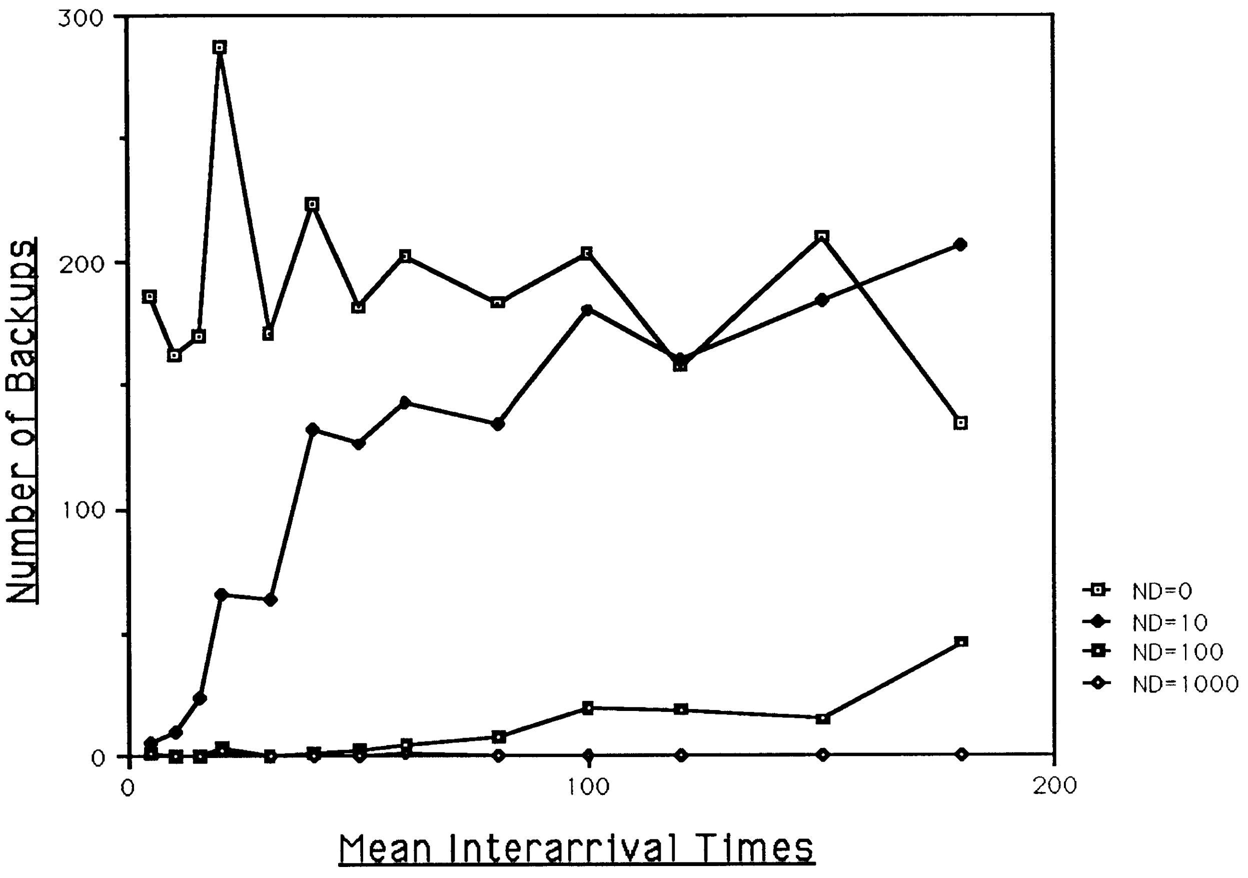


Figure 7.77 Number of Backups for Continuous Dynamic Repartitioning while varying ND=0, 10, 100, and 1000

MIT:ND is either 0 or very large. In the extreme cases the MIT has little effect.

7.11.1.1 ND=0

In this case the continuous scheme ignores the precedence classes of each task since there is no virtual time delay for processing a node. In other words, all tasks look equally attractive to a relocating processor. In this case processors tend to follow messages through the system. Therefore, many backups occur because the precedence classes are ignored.

7.11.1.2 ND=1000

In this case the continuous scheme obeys an implicit vertical partitioning that is imposed on the network by the high ND. In this case the tasks in the higher precedence classes have much lower LVRTs than the tasks in the lower precedence classes due to the high ND; therefore, processors tend to process all the tasks in the higher precedence classes before going onto the lower precedence classes. This will prevent most backups. This behavior corresponds to the optimal case described in Section **7.5.3.3**. Therefore, no backups occur because the precedence classes are obeyed.

7.11.1.3 ND=10 (the usual case) or ND=100

In these cases the MIT:ND ratio is between extremes and the variations in MIT can have a dominant effect. For low values of MIT the continuous scheme behaves more like the ND=1000 case and very few backups occur. As the MIT increases the system behaves more like the ND=0 case and many backups occur. In this case precedence classes control scheduling when the MIT is small. This clearly shows that the ratio of the MIT:ND is the important factor in determining the effect of virtual-time delays on performance.

Figure **7.78** shows the number of backups for the continuous dynamic scheme for 8, 16 and 32 processors with ND=10. Examine the graph for MIT=180 (the rightmost point). For the 8-processor case the two trials that were averaged had 19 and 21 backups, for the 16-processor case they had 48 and 58 backups, and for the 32-processors case they had 190 and 221 backups. This approximately follows a square law: if the number of processors increases by a factor of C, then the number of backups increases by a factor of C^2. Other points on Figure **7.78** also are generally consistent with the square law. The reason for this square-law effect is unclear.

7.11.2 Discussion

Our data confirm that continuous dynamic repartitioning is very sensitive to the ratio MIT:ND regardless of the partitioning method. Therefore, for sufficiently large

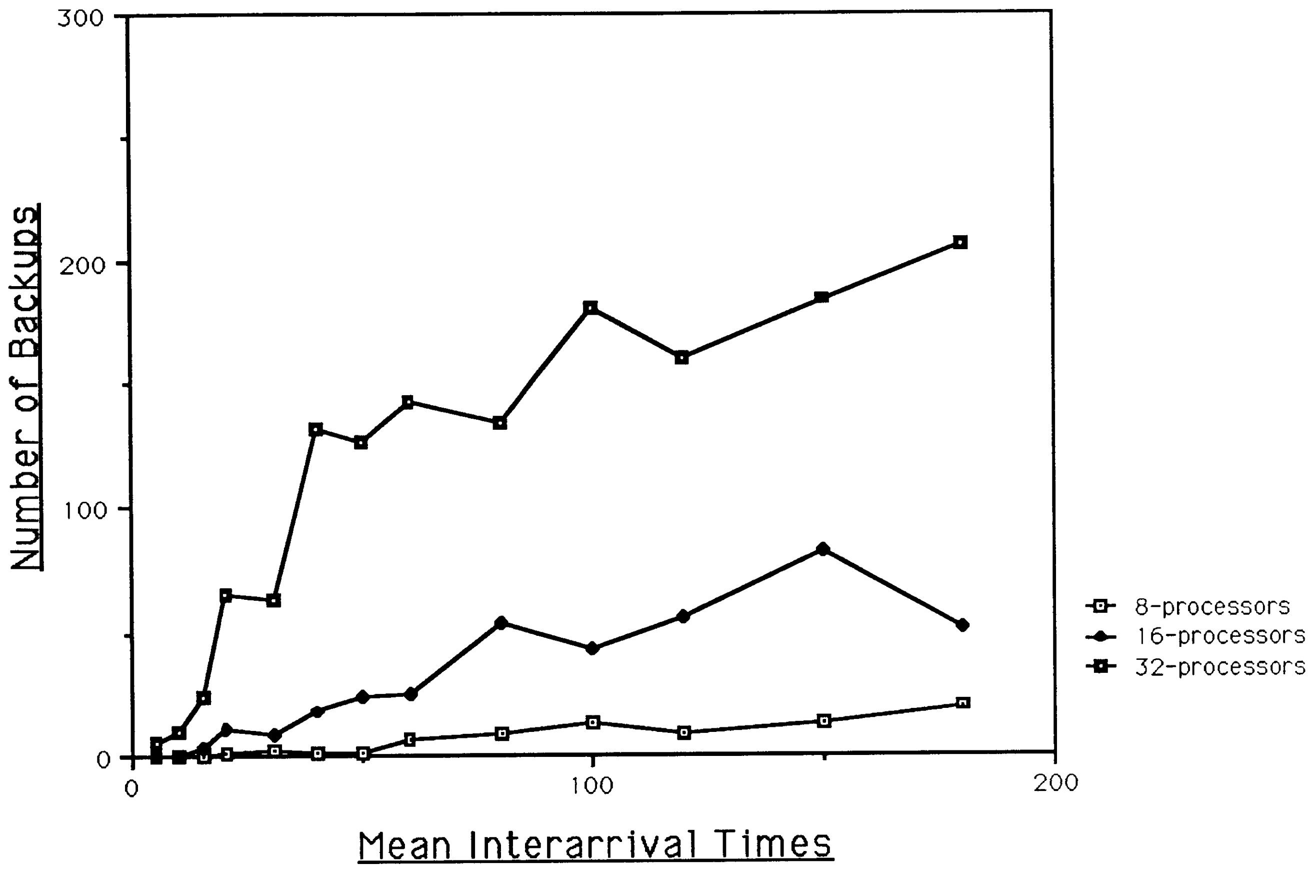


Figure 7.78 Number of Backups for Continuous Dynamic Repartitioning while varying the number of Processors = 8, 16, and 32

MIT the continuous scheme is worse than any of the other scheduling schemes when using a good partitioning method (V_6, MC_8, MC_{12}, or MC_{24}). However, in the horizontal partitioning case, continuous dynamic repartitioning looks good since all the other schemes fare so poorly.

Recall that the synchronization effect went against the precedence classes in horizontal partitioning, thus many backups occurred. On the other hand, the continuous scheme tends to synchronize the entire network to one LVT; therefore, it does not allow the adverse synchronization effect to occur in horizontal partitioning. Thus, it outperforms most of the other scheduling schemes in the horizontal partitioning case.

When the nodal delay is large, continuous dynamic repartitioning behaves well since processors will stay in the input partition as long as possible. On the other hand when ND is small, then the processors tend to follow the messages through the system until they exit the system. The second experiment (shown in Figure **7.77**) was used to confirm these beliefs.

A different experiment (shown in Figure **7.78**) shows a square-law effect in which the number of backups rises as the square of the number of processors. The reason for this square-law effect is unclear.

7.12. Group List Order Importance

This section discusses how the group-list order can affect the number of backups during both initialization and actual simulation runs. The group-list order effects are demonstrated by experiments run in the extreme experiments chapter.

The importance of the group list was discussed in Section **6.5**. For the fixed-list and circular-list dynamic repartitioning, the order of the group list played a vital role in determining the performance of the simulation. In the fixed-list strategy, the group list determined the exact order in which partitions were searched in order to find work. By placing partitions primarily consisting of tasks that should execute at the end of the simulation at the the front of the group list, we will force many extra backups. This is because these partitions will be the first to receive any relocating processors. The group list also plays a timing role in the circular-list scheme. Since this list is circular, bias toward any one partition is minimized. However, the circular-list strategy will still impose an order on choosing partitions.

In the longest task queue scheme, the order of the group list plays a large role only if there are two partitions with the same number of tasks on their task queues. In this case, the longest task queue scheme performs like the fixed-list scheme since it relocates processors to the partitions higher up on the group list. The group list comes into play whenever there is a tie in the relocation criteria used in a particular scheduling scheme. This is also true in the LVRT or EVRT schemes when there is a tie for the lowest LVRT or lowest LEVRT among the partitions.

At initialization, the group list plays another role. A partition is activated when processors start executing tasks within that partition. The order of the group list determines the order in which the partitions are initially activated. This normally does not have a large impact on simulation runs; however, for very short simulation runs where most of the work is done at the beginning of the simulation, the effect is much the same as that of the fixed-list scheme. This is because the partitions at the front of the group list are assigned processors first, so they have some priority over the other partitions, as in the fixed-list scheme. If the simulation is very short, then the partitions at the front of the list could finish all their work before some of the other partitions start. Some examples of how the order of the group list plays an important role in short simulations are shown in experiments of Figures **6.6-6.10** and are discussed in Section **6.5**.

7.13. Circuit Simulator

Finally, we discuss a four-bit adder circuit simulation. The simulation was divided into four partitions, one for each adder. We examine the effects of increasing the MIT on backups and processing time. These experiments show that the effects that appeared in the network experiments also apply for a different type of simulation.

7.13.1 Four-bit Adder

In this set of experiments a circuit simulation was run. The circuit was a four-bit adder. This circuit is described in Section **5.3** and is illustrated in Figure **5.2**. Each one-bit adder was a separate partition.

This experiment (Cir_{32}) is summarized in Figure **7.79**. In this experiment each of the four partitions started with eight processors for a total of 32. The input messages consisted of a zero or one (representing the new state of the corresponding input wire) chosen at random with equal likelihood. Each adder had two driver tasks where messages originated. The differences in VRT of successive messages was based on an exponential pdf which depended on the MIT (this does not correspond to the normal

Four Bit Adder Circuit Simulation with 32 Processors

Initial Processor Allocation	(8 8 8 8)
Effects of Increasing MIT	# of Backups increases in all strategies except for EVRT where it decreases
Scheme Ordering Based on # of Backups (ordering from worst to best)	1) Partitioning 2) Circular-list 3) Fixed-list, TQ and LVRT 4) EVRT
Lazy Message Cancellation's Effects on # of Backups	Increases the number of backups in all schemes except EVRT

Effects of Increasing MIT on Processing Time	Processing times increase for all schemes
Scheme Ordering Based on Processing Time (ordering from worst to best)	1) Partitioning and EVRT 2) Circular-list, Fixed-list, TQ and LVRT
Lazy Message Cancellation's Effect on Processing Time	Increases processing time in all schemes

Figure 7.79 Summary of Four-bit Adder with 32 Processors

operation of a 4-bit adder, since normally the adder would receive new inputs in parallel waves with many of the inputs changing at about the same time). The results of this experiment are shown in Figures **7.80** and **7.81**. Figure **7.80** shows the number of backups for each of the scheduling policies as the MIT increases from 5 to 60. The different nodal delays were arbitrarily chosen as:

1. 2 virtual-time units for inverters
2. 3 virtual-time units for and-gates

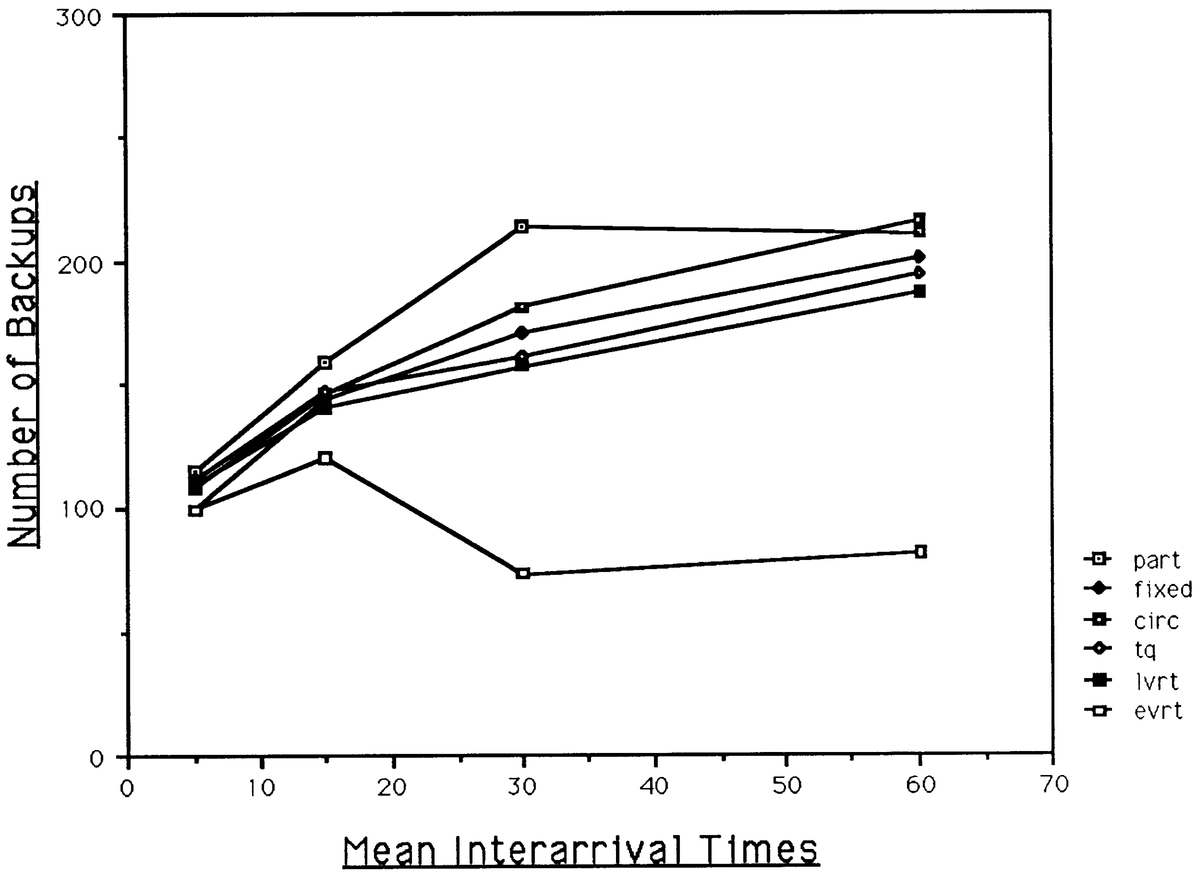


Figure 7.80 Number of Backups versus MIT for Cir_{32}

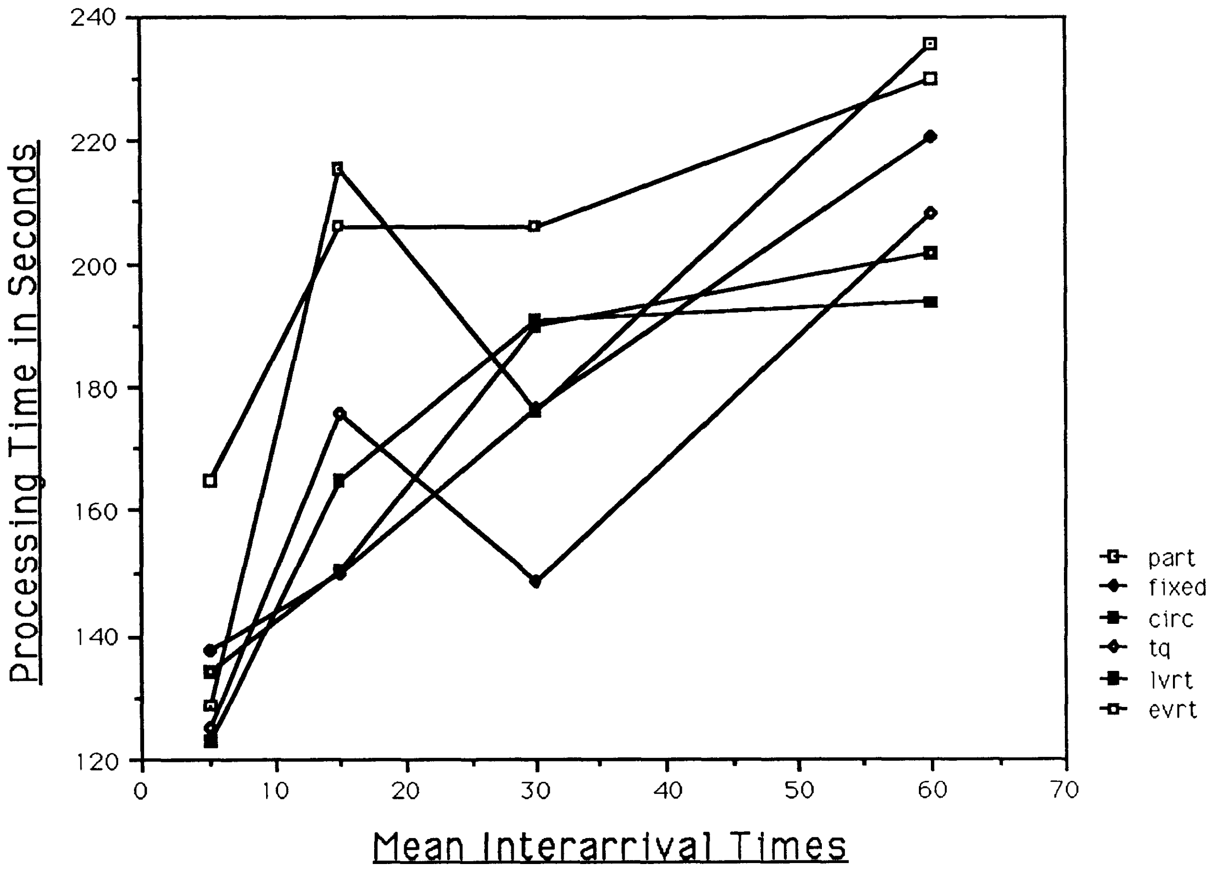


Figure 7.81 Processing Time versus MIT for Cir_{32}

3. 5 virtual-time units for or-gates.

As the MIT increases, backups increase for all the different scheduling policies, except for the EVRT strategy. The EVRT strategy has the fewest backups, whereas static partitioning has the most backups.

Lazy message cancellation *increases* the number of backups in all schemes except for the EVRT scheme. This effect is shown in Figure **7.83** for the fixed-list strategy and **7.84** for EVRT. This is the opposite effect that lazy message cancellation had in the butterfly network example. Figure **7.84** shows that the lazy message cancellation had no systematic effect on the EVRT scheme.

Figure **7.81** shows the processing times for each of the scheduling policies as the MIT increases. As the MIT increases, processing times increase for all the schemes. The

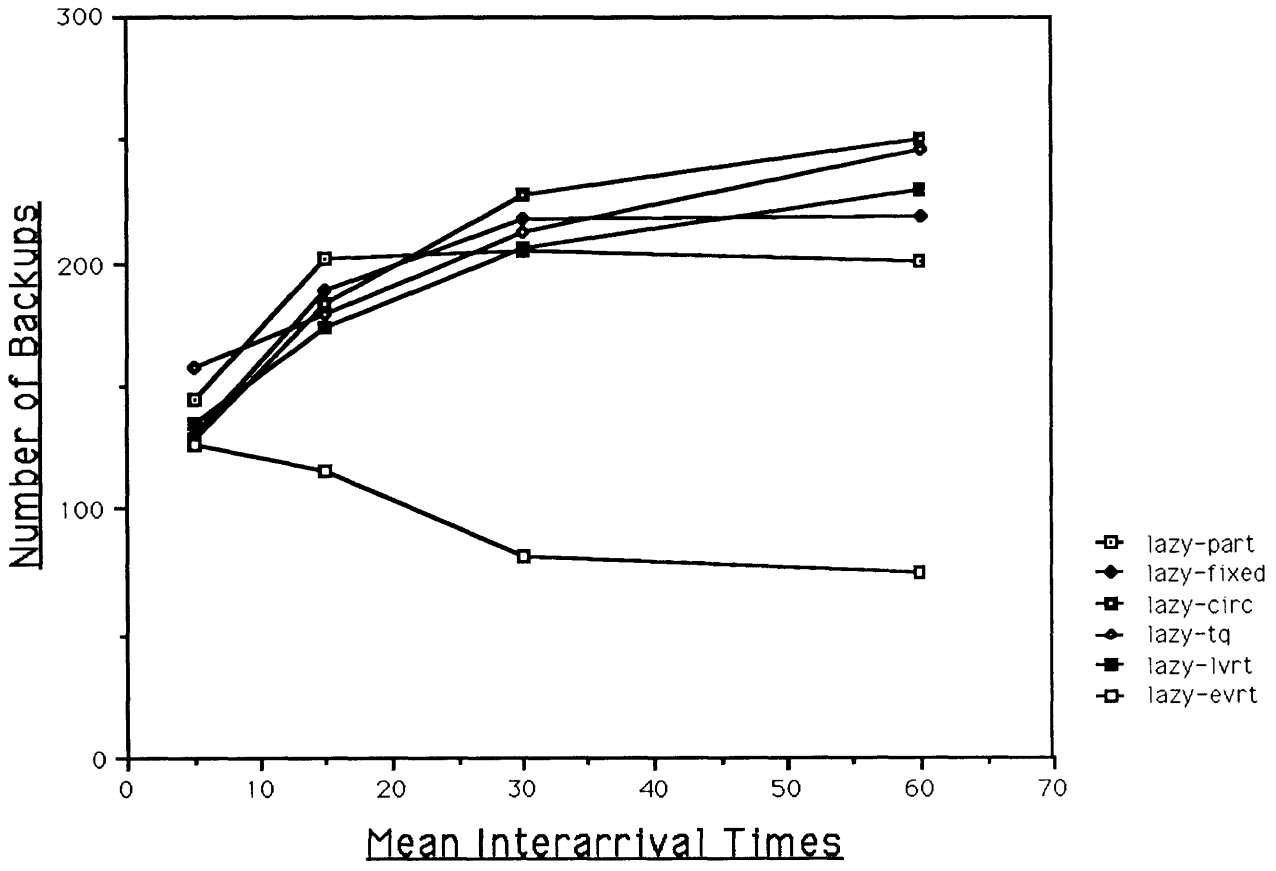


Figure 7.82 Number of Backups versus MIT for Lazy Cancellation for Cir_{32}

EVRT and static partitioning schemes perform the worst, whereas the other schemes perform about the same.

Lazy message cancellation increases the processing times for all schemes and is illustrated in Figure **7.85**. This effect is shown for the fixed-list strategy in Figure **7.86**.

7.13.2 Discussion

This example clearly shows that some of the prominent effects in the network simulation also apply to the circuit simulation. The virtual-time delays (increasing MITs) increased the number of backups and the processing times. For this simulation the nodal delay varies for different circuit elements. The ratio MIT:ND still plays

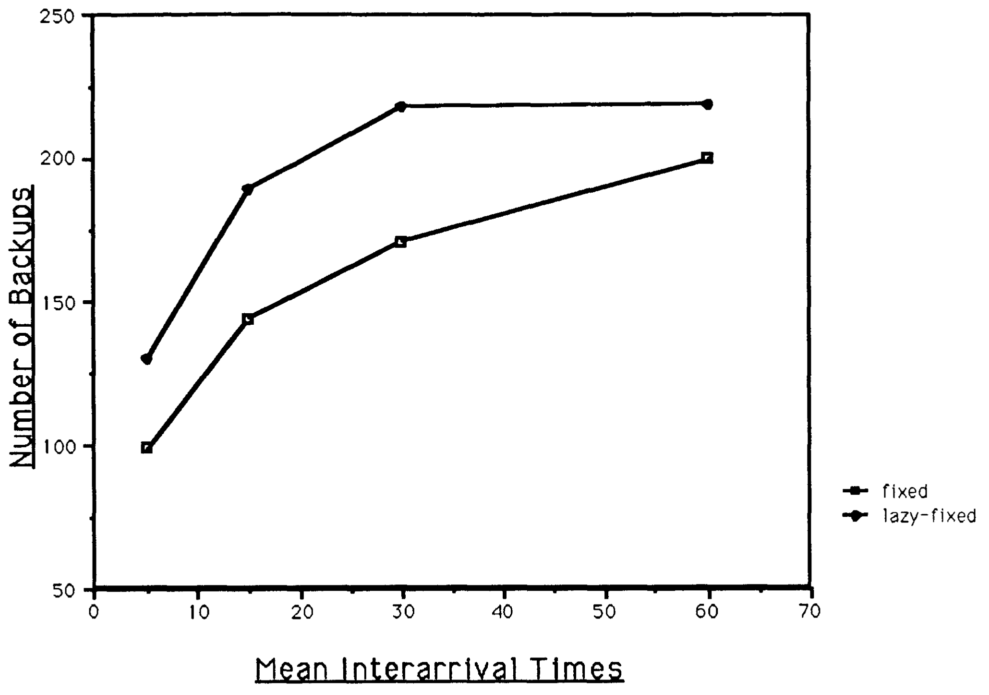


Figure 7.83 Number of Backups versus MIT for Cir_{32} using Aggressive or Lazy Cancellation with Fixed-list scheduling

an important role in simulation performance, but in this case it is more complicated because there are three different nodal delays.

Unlike the network simulation, each task in the circuit simulation has state. For example, each wire element maintains the signal value of the wire and each circuit element maintains the signal value or signal values of its inputs. In the network simulation, each node's output message is dependent only on its input message (when there are no conflict delays). On the other hand in the circuit simulation, each task's output message is dependent on its input message and the state of its circuit element. Let us call this the ***state attribute*** of the circuit simulation.

A peculiarity of the circuit simulation is that lazy message cancellation seems to worsen the performance. We hypothesize that lazy message cancellation delays backups (and allows further progress) of tasks that should have been backed up, thus causing additional backups when those tasks finally are backed up.

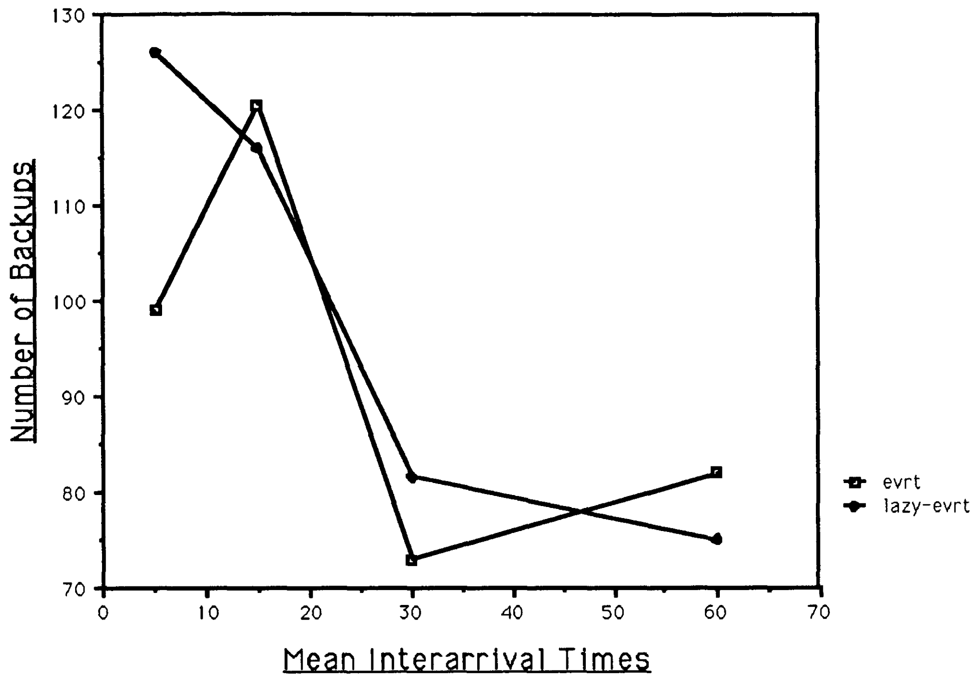


Figure 7.84 Number of Backups versus MIT for Cir_{32} using Aggressive or Lazy Cancellation with EVRT scheduling

One might think that lazy message cancellation has the following advantages over the aggressive message cancellation policy:

1. fewer anti-messages, and
2. fewer secondary rollbacks, which are additional backups caused by anti-messages and reprocessed preempted messages (preempted messages are those messages that have already been processed with greater virtual time than the preempting message's VRT).

On the other hand lazy message cancellation delays needed backups, which could increase the total amount of backup and *increase* anti-messages and secondary rollbacks.

Generally, lazy message cancellation requires more processing time at the backed-up node than aggressive message cancellation, since every reprocessed preempted message must be checked to see if the same message was already sent. Therefore, if lazy

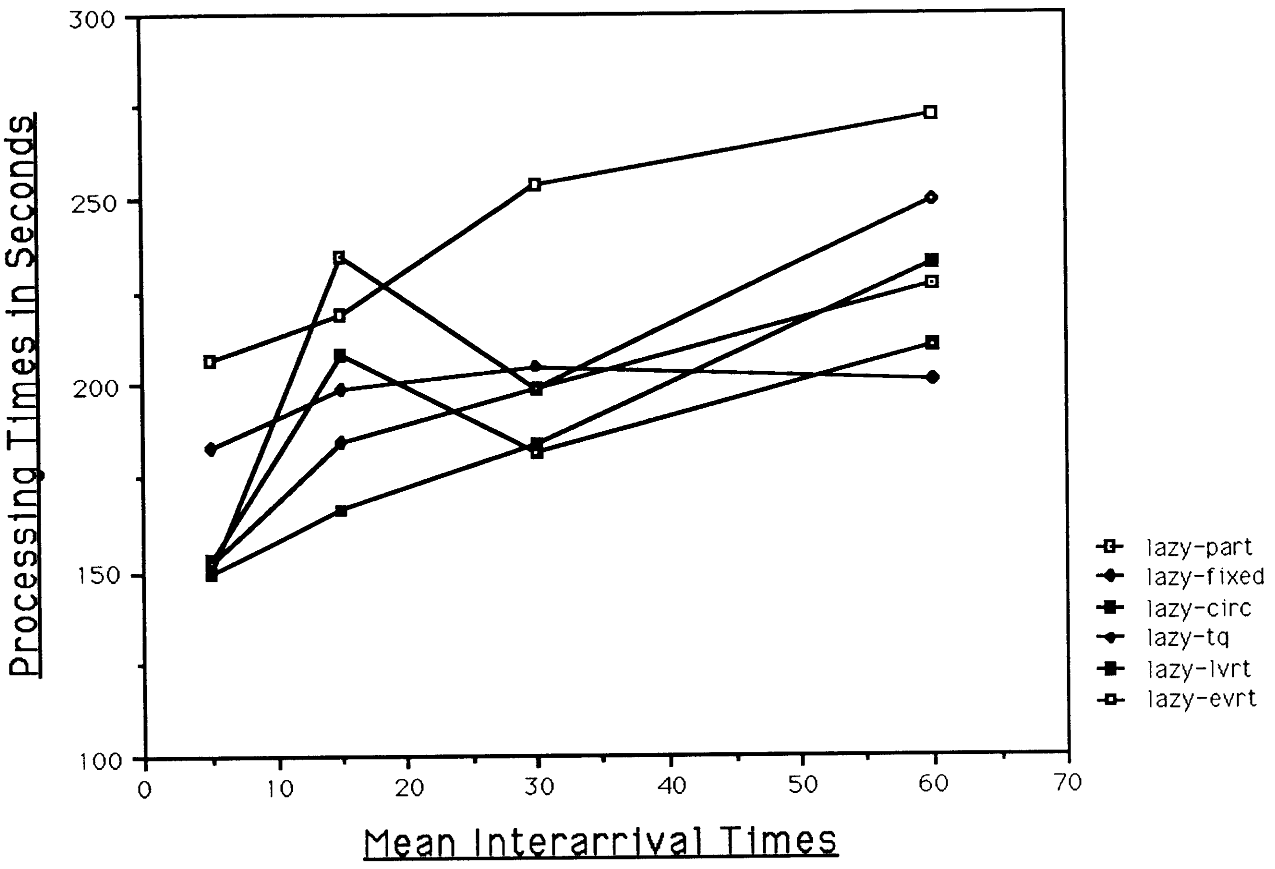


Figure 7.85 Processing Time versus MIT for Lazy Cancellation for Cir_{32}

message cancellation must send out the same number of anti-messages and reprocessed preempted messages as the aggressive case, then the lazy case will take more processing time.

One potential reason why the lazy message cancellation mechanism performed worse than the aggressive case is due to the ***state attribute*** of the circuit simulation. When a backup occurred in the network simulation, the preempted messages usually remained the same when they were reprocessed (except when VRTs change due to conflict delays). Therefore, when using lazy message cancellation in the network simulation, the backed-up node would send almost no anti-messages and reprocessed preempted messages. Thus, the processing time for lazy message cancellation would be much less than for the aggressive case if there were many backups. Furthermore, these few anti-

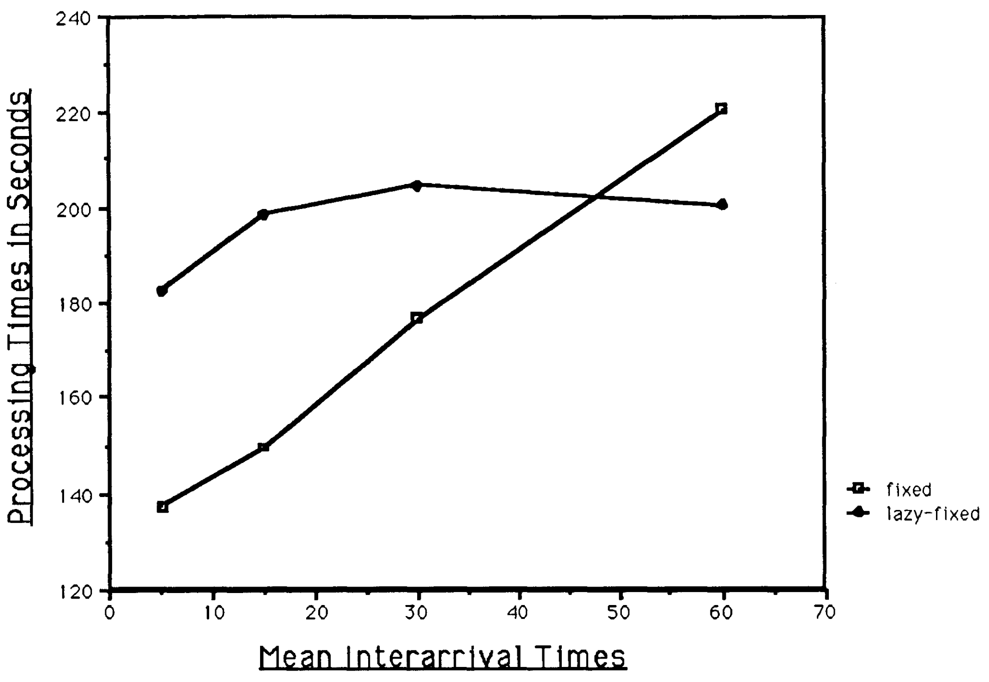


Figure 7.86 Processing Time versus MIT for Cir_{32} using Aggressive or Lazy Cancellation with Fixed-list scheduling

messages and reprocessed preempted messages caused very few secondary rollbacks. The key point here is that since there were very few anti-messages and reprocessed preempted messages, lazy message cancellation did not delay needed backups.

On the other hand, in the circuit simulation, many of these preempted messages will change after they are reprocessed (because of backup), due to the *state attribute* of the circuit simulation. All of these messages that have changed must be canceled (via anti-messages) and the new reprocessed messages must be sent. The problem with lazy message cancellation is that these anti-messages and reprocessed preempted messages are only sent when absolutely necessary. Therefore, there could be a large real-time delay before these anti-messages and reprocessed preempted messages are sent. Real-time delays have already been shown to cause additional backups, but the key point here is that lazy message cancellation may hold up needed backups, thus causing an increase in the total number of backups in the system. These additional backups and the additional inherent processing time cost of the lazy message cancellation could easily cause the lazy case to take more processing time than the aggressive case.

In this simulation we conclude that lazy message cancellation delays backups of tasks that should have been backed up, thus causing additional backups. That lazy message cancellation causes more backups means that we cannot blindly choose lazy message cancellation for all simulations. Instead, we should carefully study the total number of messages processed in either the aggressive or lazy message cancellation techniques.

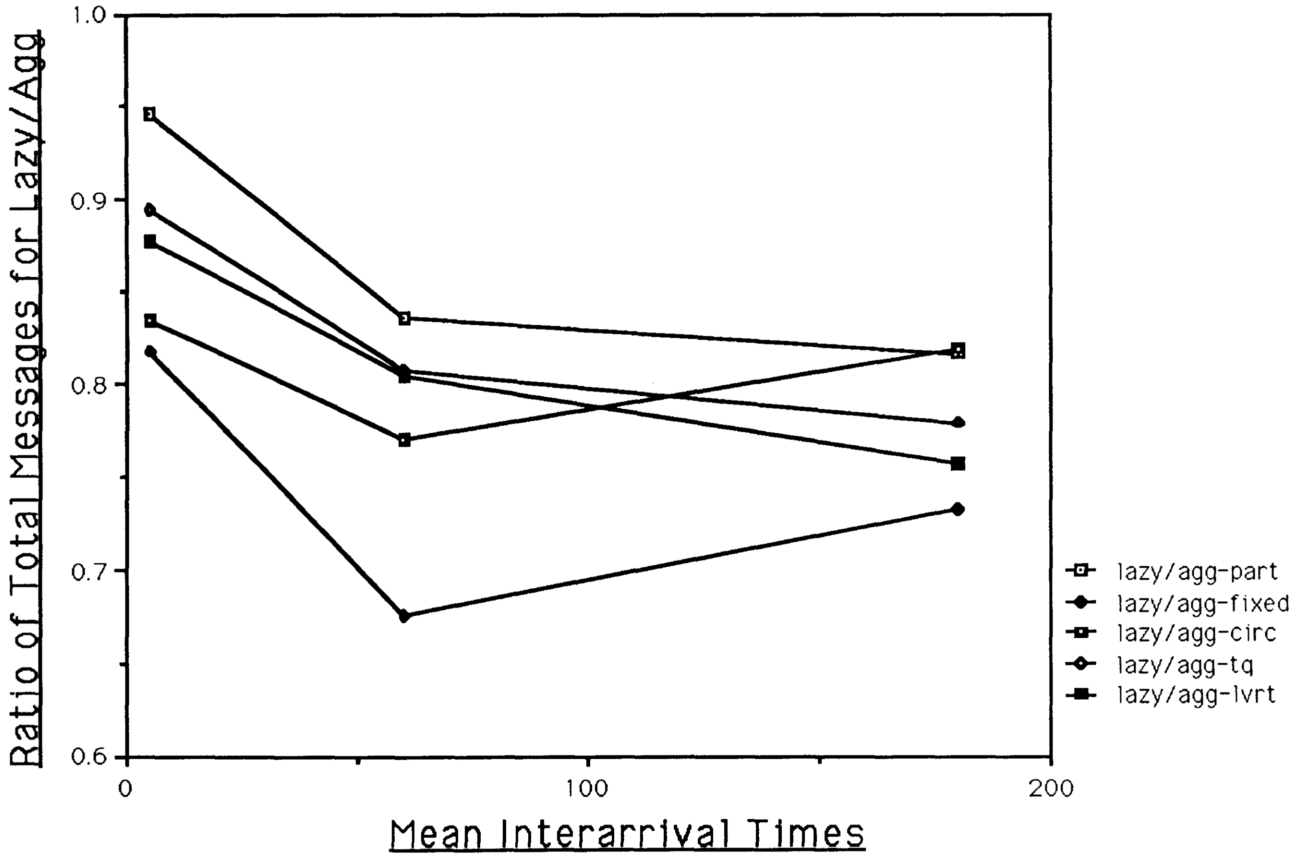


Figure 7.87 The Ratio of Total Messages for Lazy/Aggressive Cancellation on a network simulation H_{16} versus MIT

One might conjecture that since the lazy message cancellation mechanism only sends anti-messages when absolutely necessary, the total number of messages for the

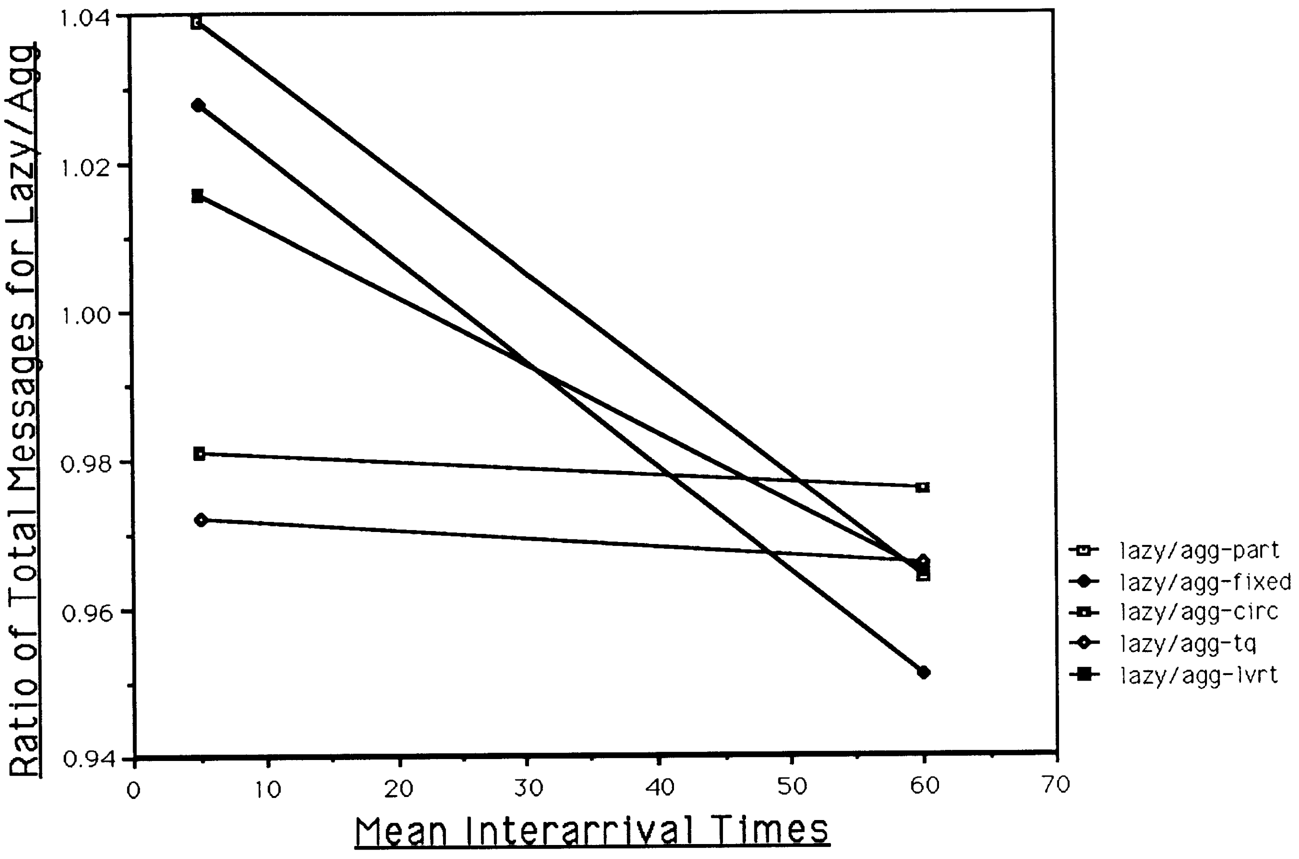


Figure 7.88 The Ratio of Total Messages for Lazy/Aggressive Cancellation on a circuit simulation Cir_{32} versus MIT

lazy mechanism is the minimum number of messages required to execute the simulation. This is not correct (as is seen in Figure **7.88**)! Since lazy message cancellation can delay needed backups, the end result could be more backups and even more messages and anti-messages.

In any case, the total number of messages processed is a good performance measure of message cancellation techniques because a smaller number of total messages processed usually translates to fewer backups and anti-messages. Therefore, the ratio of total number of messages in the lazy mechanism versus the total number of messages in the aggressive mechanism will give a good measure of which message cancellation mechanism is better. Figure **7.87** shows this ratio of

$$\frac{Total\ number\ of\ messages\ for\ Lazy\ cancellation}{Total\ number\ of\ messages\ for\ Aggressive\ cancellation}$$

for the network simulation as the MIT is increased. In the network simulation the ratio varies between 0.68 and 0.95 meaning that many of the anti-messages in the network simulation are probably not needed. On the other hand Figure **7.88** shows this ratio of

$$\frac{Total\ number\ of\ messages\ for\ Lazy\ cancellation}{Total\ number\ of\ messages\ for\ Aggressive\ cancellation}$$

for the circuit simulation. In the circuit simulation the ratio varies between 0.95 and 1.04 which means that the number of messages for the lazy mechanism is about the same as for the aggressive mechanism.

As we have seen in this section, lazy message cancellation is not a panacea. In fact, in some cases of the circuit simulation, the lazy mechanism required more backups than the aggressive mechanism to complete a simulation.

The key difference between the circuit and the network simulations is state. In the network simulation, a task's output messages depended only on the previous input messages (except for conflict delays), whereas in the circuit simulation the output messages depended on the state of the task. Therefore, when backup occurred in the network simulation while using the lazy cancellation mechanism, many of the preempted messages never needed to be canceled and resent. On the other hand, in the circuit simulation a preempting message would usually change the state of the preempted task, thus requiring cancellation of the preempted message.

Lazy message cancellation is good when the simulation depends only weakly on state because the preempted messages will usually not be canceled and therefore we do not delay needed backups. On the other hand if the simulation depends strongly on state, delaying the cancellation of preempted messages could easily delay needed backups which could in turn cause more backups. Therefore, the lazy mechanism is more likely to delay needed backups in a simulation that depends strongly on state.

In the network simulation, the lazy message cancellation mechanism usually had fewer total messages than the aggressive mechanism. This explains why the lazy mechanism greatly improved the performance of the network simulation under horizontal partitioning, whose many backups accentuated the difference between lazy and aggressive message cancellation. On the other hand, in the circuit simulation the total number of messages is almost the same. This explains why the lazy mechanism did not improve the processing time performance in the circuit simulation. Furthermore, since the lazy mechanism requires more processing per message, it will usually take more processing time than the aggressive mechanism if both require the same total number of messages.

7.14. Speedup

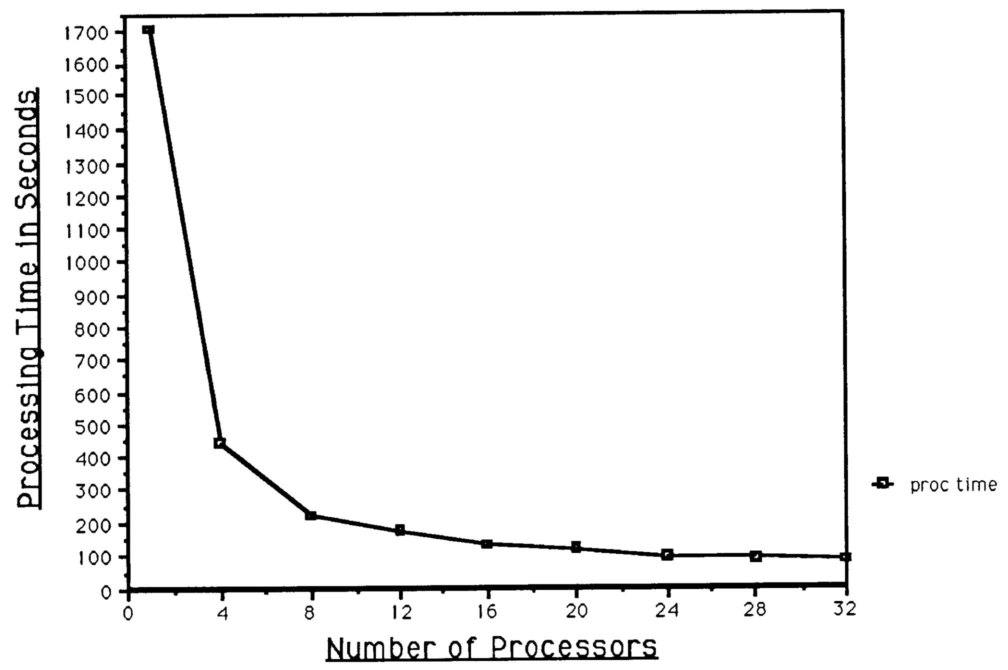


Figure 7.89 Processing Time versus Number of Processors

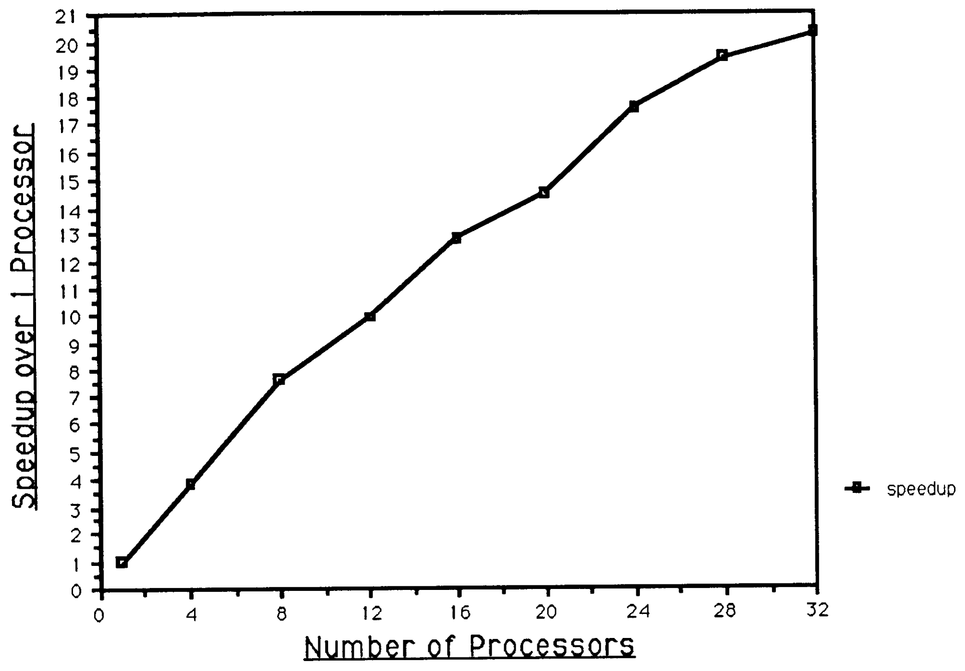


Figure 7.90 Speedup versus Number of Processors

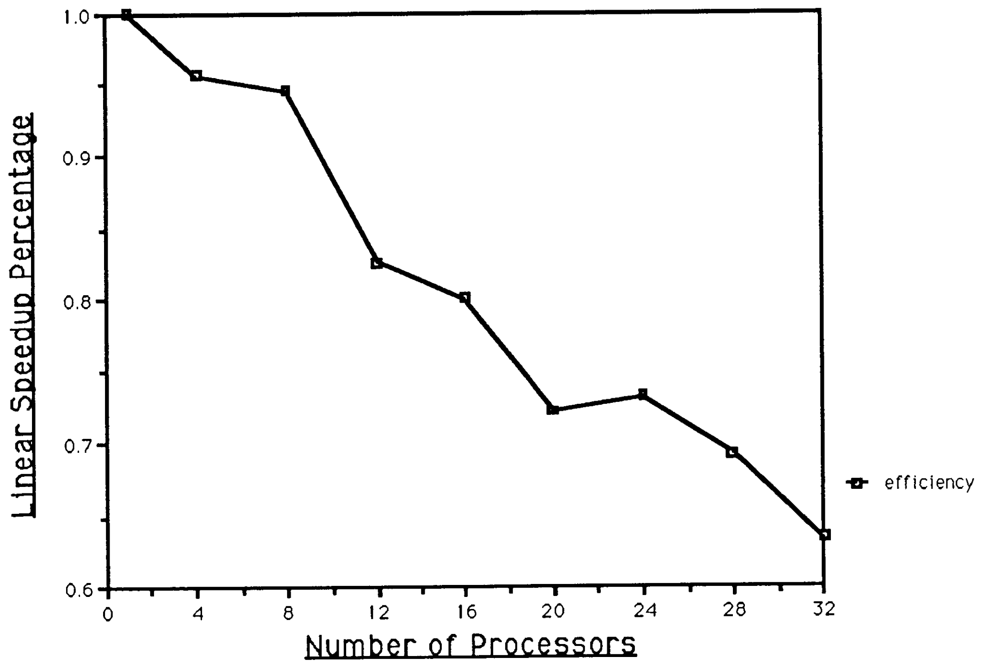


Figure 7.91 Percentage of Linear Speedup Efficiency versus Number of Processors

As a last experiment, we show the speedup of our simulation system as the number of processors increases. In this experiment we used vertical partitioning, the LVRT scheduling scheme and only 10 messages per input node (this was to reduce the amount of processing time for the 1 processor case). In Figure **7.89** we show the amount of processing time for a simulation run as the number of processors varied. In Figure **7.90** we show the amount of speedup for a simulation run as the number of processors varied. Here we define speedup for a simulation with n processors to be the processing time with 1 processor divided by the processing time with n processors. Figure **7.91** shows the *efficiency* – the speedup divided by the number of processors. Figure **7.91** shows that the speedup starts up almost linear (efficiency .95 for 4 processors) and then decreases (efficiency .63 for 32 processors). Increasing the number of processors after a certain point will not speed up the simulation. Let us call this point processor saturation.

We hypothesize that the efficiency decreases as the number of processors increases because there may never be as many tasks to process as processors at any time during the simulation. Also, the amount of parallelism in a simulation may vary greatly from the start to the end of the simulation. Assuming we are using n processors, at a time when the simulation has less than n tasks to process, the extra processors will be suspended. This suspended or idle time is included in the processing time as discussed in Section **6.2.**.

7.15. Summary

In this chapter, we discuss various results which help one to predict the performance of a simulation based on the properties of the simulation. These results were used to confirm our theories. Experimental results were based on varying the order of the group list, partitioning methods, scheduling policy, lazy versus aggressive message cancellation, message queue lengths, and virtual and real time delays.

7.15.1 Lazy versus Aggressive Message Cancellation

Lazy message cancellation only benefited the system when a poor partitioning method was used, such as horizontal or random partitioning in the network simulation. This was because aggressive message cancellation compounded the backups in what we term the "aggressive backup effect." The aggressive backup effect was prominent in the horizontal and random partitioning methods in the network simulation; therefore, lazy message cancellation performed up to 50% better in these cases. On the other hand, in the V_6, MC_8, MC_{12} and MC_{24} cases, the difference between the performance of lazy and aggressive message cancellation was negligible because of the absence of the aggressive backup effect. In the 4-bit adder case, the performance of lazy message cancellation was worse than that of aggressive message cancellation.

7.15.2 Message Queue Length

There is an interesting relationship between task backups and the input message queue length at the task. Longer message queue lengths were found to be associated with fewer backups, because a large message backlog allows messages to be sorted in virtual time order. As the message queue lengths increased, the Time Warp method looked more like the safe case of the Network Paradigm method.

7.15.3 Best Partitioning

Figure **7.92** shows a comparison of the best scheduling schemes, judged by their number of backups, for each partitioning method when used with the 16-input-node butterfly network. The LVRT scheme on V_6 and static partitioning on MC_{24} performed the best. The second tier consists of static partitioning on MC_{12} and MC_8. These same two tiers exist in the lazy message cancellation case, shown in Figure **7.93**. Figure **7.94** shows a comparison of the best scheduling schemes, in terms of processing time, for each partitioning method. The LVRT scheme on MC_8, the static partitioning scheme on MC_{24}, and the fixed-list scheme on MC_{12} performed the best. The same ordering appears in the lazy message cancellation case, shown in Figure **7.95**.

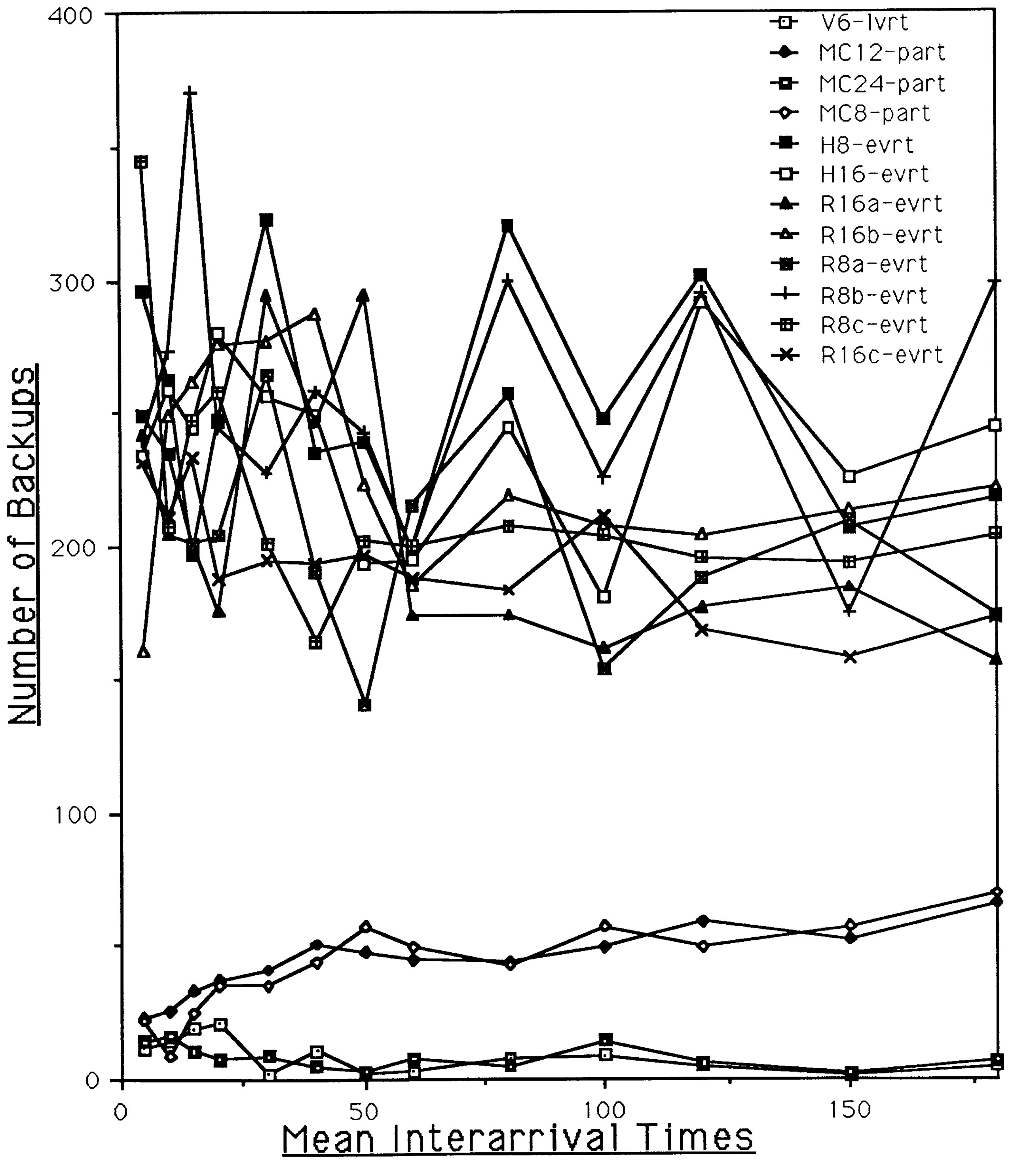


Figure 7.92 Best-Case Number of Backups vs MIT in Aggressive Message Cancellation

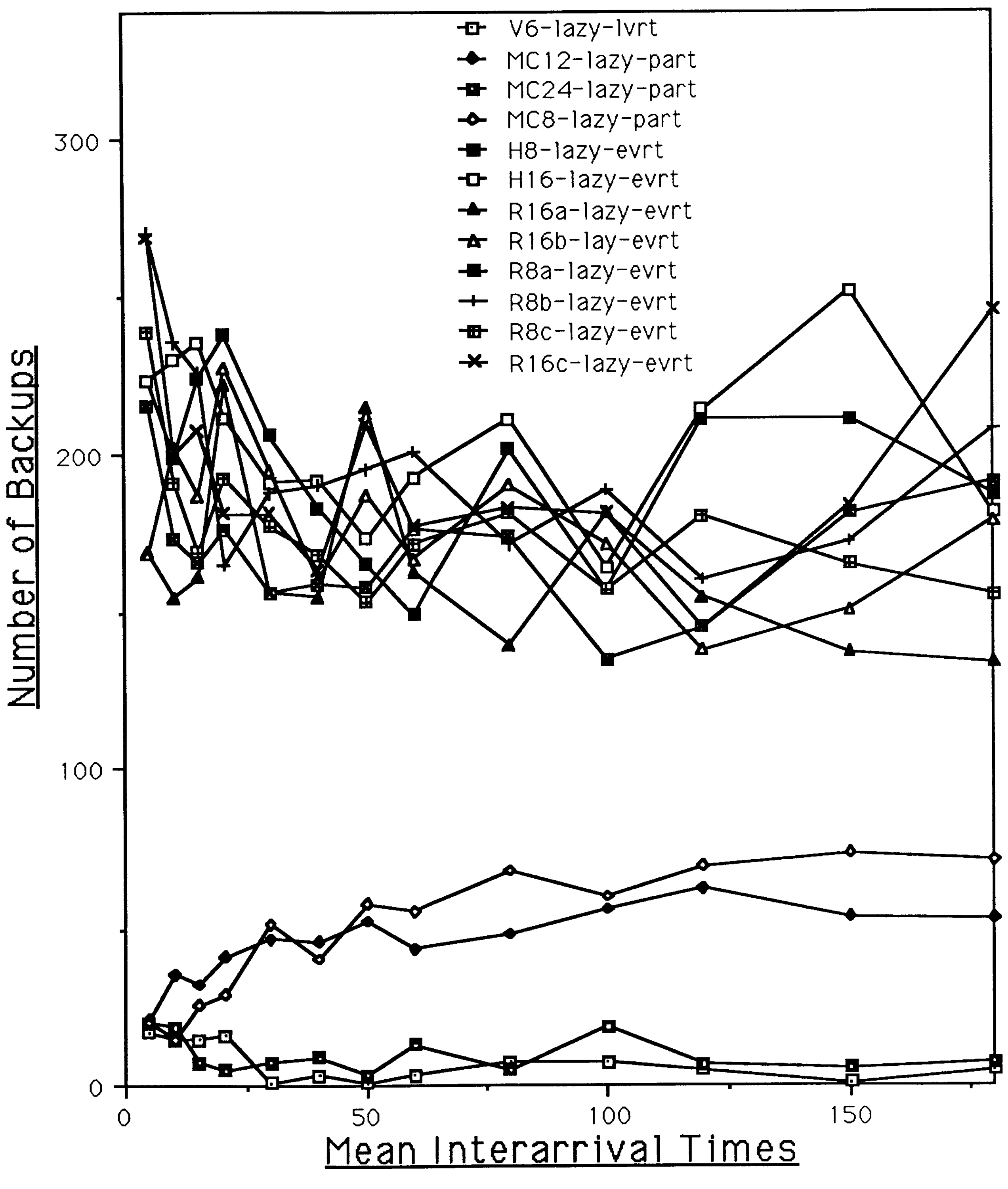


Figure 7.93 Best-Case Number of Backups vs MIT in Lazy Message Cancellation

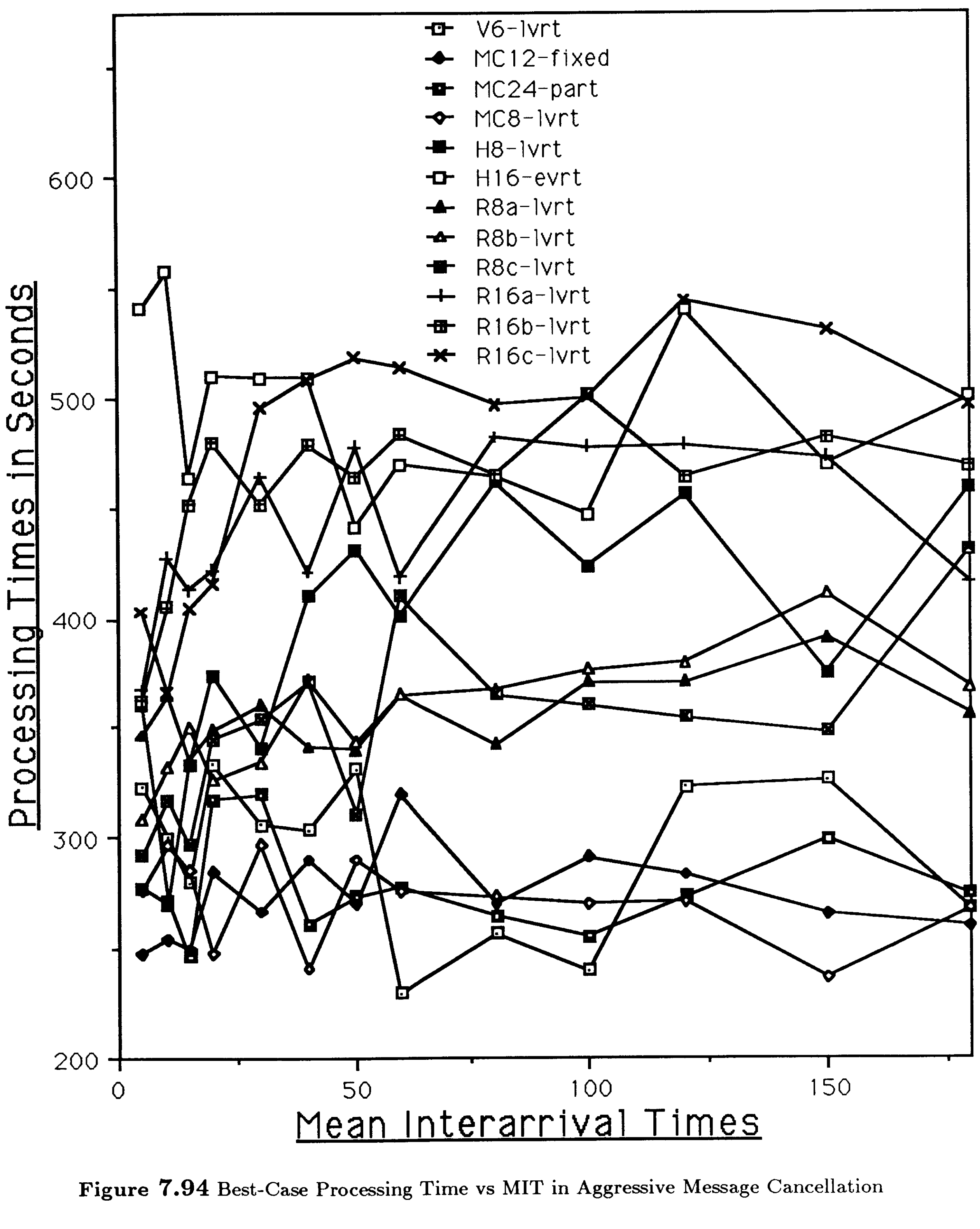


Figure 7.94 Best-Case Processing Time vs MIT in Aggressive Message Cancellation

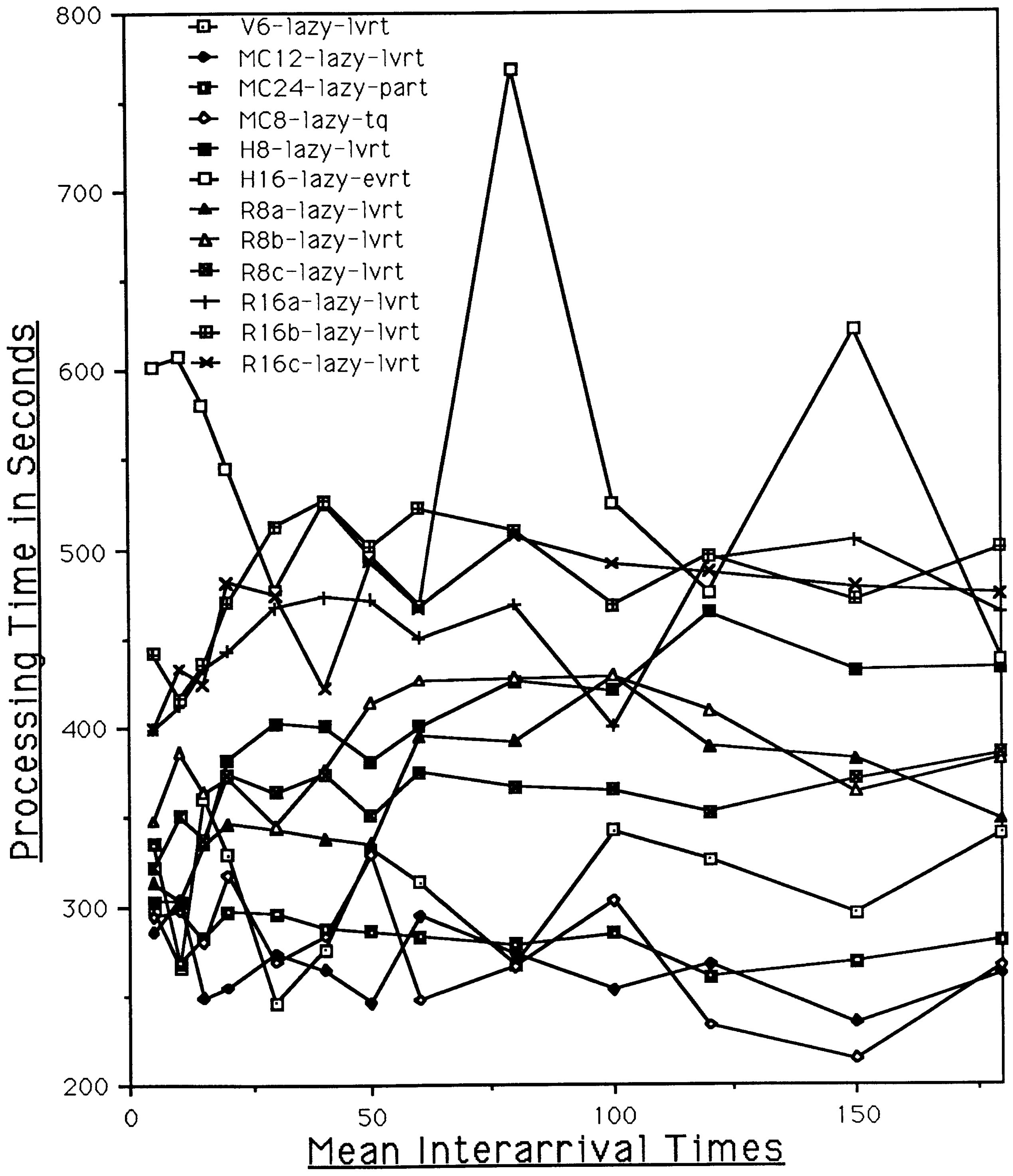


Figure 7.95 Best-Case Number of Processing Time vs MIT in Lazy Message Cancellation

It is not always easy to identify the type of special purpose partitioning and scheduling scheme that will run best for a particular application. As far as choosing a method to partition the network simulation, minimum communication partitioning and vertical partitioning were always found to be clearly better than either horizontal or random partitioning.

7.15.4 Scheduling Schemes

In the H_8 case the EVRT scheme had fewer backups than the other schemes, while in the H_{16} case the EVRT scheme had both fewer backups and lower processing time than the other schemes. In general the performance of the scheduling schemes under the H_8 and H_{16} partitioning methods tended to be much worse than that under minimum communications or vertical partitioning. In R_{8a}, R_{8b}, R_{8c}, R_{16a}, R_{16b} or R_{16c} partitioning, the EVRT method had significantly fewer backups than did the other scheduling schemes.

Optimizations of scheduling schemes are very useful when the partitioning method is flawed. However, when the partitioning method is good, the additional processing time needed for optimizations can worsen a simulation's performance. Given a good partitioning method, such as minimum communications, there is no benefit of using a complicated scheduling scheme such as the EVRT scheme. In fact, the EVRT scheme actually *increased* backups in many cases, such as minimum communications or vertical partitioning (see Figures **7.60**, **7.61**, **7.62**, and **7.12**). When the partitioning is bad, as in random or horizontal partitioning, then the distinctions between the different scheduling schemes and message cancellation policies become greater. For instance, even though the EVRT scheme averages 80 more backups than any other scheme for V_6, it is not nearly as bad as choosing the fixed-list scheme for H_{16} which averages 900 more backups than the EVRT scheme.

We found that the simple static partitioning scheme was best for partitioning methods that were susceptible to partition saturation, such as MC_8, MC_{12}, and MC_{24}. The EVRT scheme was best for partitioning methods that suffered from the aggressive backup effect, such as H_8, H_{16}, R_{8a}, R_{8b}, R_{8c}, R_{16a}, R_{16b}, and R_{16c}, whereas the LVRT scheme was best for partitioning methods that did not suffer from either partition saturation or the aggressive backup effect, such as V_6.

By examining Figures **7.92**, **7.93**, **7.94**, and **7.95**, it appeared that without prior special knowledge of the simulation, one should choose the EVRT scheme in order to minimize backups, or the LVRT scheme in order to minimize processing time for reasonable partitionings and message sequences with large MITs. The EVRT scheme minimizes backups because the EVRT scheme had the best worst-case potential. In other words the EVRT scheme had the fewest number of backups in its worst case scenarios compared to the worst case scenarios of the other schemes. Similarly, the LVRT scheme should be used to minimize processing times because it has the best worst-case potential.

Continuous dynamic repartitioning performance was found to be very dependent on the MIT:ND ratio. As the MITs increased or ND decreased, the performance of the simulation system degenerated rapidly, having many more backups and much longer processing times than any other scheduling scheme. Therefore it appears that the continuous dynamic strategy should be bypassed except for investigative purposes.

7.15.5 Times–Virtual and Real

The ratios of MIT:ND and IM:PT have dramatic effect on the performance of a simulation. In our experiments we varied the mean interarrival time MIT and we varied the inter-message real-time delay IM via artificially introduced real-time delays.

The results depicted in Figures **7.7**, **7.18**, **7.9**, **7.20**, **7.48**, **7.52**, **7.60**, **7.61**, and **7.62** show all of the scheduling methods using both aggressive and lazy message cancellation on the various partitioning methods. All of these graphs show that an increase in the MIT in virtual time causes more backups. Figures **7.8**, **7.10**, **7.49**, and **7.53** show that an increase in the MIT causes increasing processing times.

Simulation performance was also affected by the MIT:ND ratio. In our simulations we varied this ratio by varying the MIT and by varying the nodal delay ND.

In both cases, increasing the MIT or decreasing the ND, caused more backups and increased processing times. On the other hand, the EVRT scheme was unaffected by the increasing MITs because the differential, LVRT $-$ EVRT, would remain constant while the interarrival times grew. Thus the number of backups for the EVRT scheme would not increase, even for large interarrival times. The EVRT scheme may have had poor performance for low interarrival times, but for high interarrival times it always compared more favorably to the other scheduling schemes.

Driver delays or real time delays also cause more backups. Figures **7.30 - 7.32** show that the longer the driver delay in real time, the more backups. Longer delays between input messages to the system cause fewer messages in each task's message queue, which in turn cause more backups.

Real-time delays led to more backups than in simulations with no real-time delays. Longer real-time delays led to shorter average message queue lengths. The most important factor was the $\frac{IM}{PT}$ ratio. As the $\frac{IM}{PT}$ ratio grew, the number of backups increased. There was usually a threshold after which the number of backups increased quickly to a maximum.

In our experiments, we varied the ratio $\frac{IM}{PT}$ using two methods. The first method varied the IM by artificially introducing real-time delays between successive messages. The second method started with fewer initial messages and introduced a feedback loop. The feedback loop maintains the same number of messages into the input nodes as the corresponding non-feedback experiment (without real-time delays) while increasing the IM. We could not vary PT because we could not directly affect the processing times on Concert.

VIII. CONCLUSIONS

8.1. Review of Goals

Our first goal was to construct in detail a discrete event-based concurrent simulation system. Therefore, writing in Multilisp on the 32 processor Concert multiprocessor, we wrote one of the first implementations of Jefferson's Time Warp system. The essential Time Warp mechanism was combined with different scheduling strategies, to form a simulation-independent system. Our entire system consists of the simulation-independent system, various simulation-dependent modules, and modules to interface the simulation-independent and simulation-dependent systems.

Our second goal was to understand the relationship between parameters and performance. Major parameters investigated were:

1. Scheduling Policies
2. Partitioning Methods
3. Synchronization Recovery Methods (lazy vs. aggressive message cancellation)
4. Message Queue Lengths
5. Time Delays (real and virtual)

We analyzed performance in two ways:

1. Processing time
2. Number of backups.

Backups were used as a measure of performance because Concert's processing times were highly variable. They varied because garbage collection occurred at unpredictable moments.

We felt that the number of backups was a valid performance measure because backups were relatively cheap in our simulations. They were cheap because each task's environment consisted of only one variable. If one were to increase the number of variables in the environment, backups would become expensive and could influence processing times.

In this thesis we mainly worked with two types of simulations. Our goal was to try to study the most representative set of simulations in terms of:

1. State (stateless versus state)
2. Cycles (cyclic versus acyclic)

We decided to study the following two simulations:

1. The butterfly network simulation with 8 input nodes and 16 input nodes which was introduced in Section **5.4** and
2. The 4-bit adder simulation which was introduced in Section **5.3**.

Both simulations were acyclic; however, the first simulation was easily modified to contain cycles by feeding the output of the simulation back to the input nodes. Our original goal was to look at two simulations, one with and one without state. Both simulations have state; however, the state of the network simulation is only used in the case of conflicts.

Conflicts occur when two messages arriving at the same node have nearly equal virtual receive times (VRTs). Specifically, a conflict occurs if the difference between the VRTs of two messages arriving at a node is less than a given *conflict delay.* Conflict delay represents the amount of virtual time for a task to process a message. Therefore, a conflict occurs when (in virtual time) a task receives a message, m_1, while the task is processing another message, m_2.

When a conflict occurs, the task cannot process m_1 in virtual time until the task completes processing m_2. Therefore, the VRT of m_1's corresponding output message will include the time for all conflict messages to complete processing as well as one conflict delay.

Therefore, network simulations that have a small chance of conflicts may perform like a stateless simulation. This will occur when the virtual-time delay between input messages (MIT) is high relative to the conflict delay. Conflict delays are discussed in

Section **5.4**.

These two simulations represent acyclic simulations well. The 4-bit adder simulation shows an example of simulations that are very dependent on state and have tasks with many different functions. On the other hand the network simulation shows an example of simulations that are not very dependent on state and have only two different functions for tasks. The 4-bit adder is representative of digital circuits (without cycles). The butterfly network is representative of Fast Fourier Transforms, processor-memory access paths for certain computer architectures, and computer networks.

8.2. What have We Learned?

Time Warp simulation systems consist of tasks, which are assigned to processors by a scheduler. Each task has an LVRT, which is the the virtual receive time (VRT) of the first message on its input message queue. The scheduler schedules the tasks via a task queue, which is ordered in terms of task LVRTs (the lower the LVRT the higher the priority). The scheduler assigns a task to a processor depending on the priority of the task on the task queue. This is called the non-partitioning case in our system.

When backup occurs at a task in the Time Warp system, all messages that were processed by the task after the VRT of the preempting message are canceled immediately by the sending of anti-messages. This is called the aggressive message cancellation policy. In our system we used both this policy and a lazy message cancellation policy which canceled messages only when absolutely necessary.

In our system, we implemented partitioning, which grouped tasks together as well as distributed the task queue where processors chose tasks. The way in which we partitioned a set of tasks was called the partitioning method. At any given time, each processor is assigned to a partition and may only process tasks in that partition.

As one might expect, distributing the task queue decreased the waiting time of processors accessing the task queue. However, partitioning can increase the number of backups in a simulation. When there is no partitioning, all tasks have similar LVT's due to synchronization; however, when partitioning is used only tasks within partitions have their LVT's synchronized. Therefore, inter-partition messages can cause backups because each partition is synchronized to a different LVRT (a partition's LVRT is the lowest LVRT of all its tasks).

The scheduling scheme determines if and when processors relocate to other partitions. In the static partitioning scheme, processors remain in their originally assigned partition. In dynamic scheduling schemes, however, a processor relocates to another partition if there is no work in its current partition.

We contrasted the static partitioning scheme to many varieties of dynamic scheduling schemes. We expected the dynamic scheduling schemes to outperform the static scheduling scheme because dynamic scheduling does not waste processors on partitions where there is little or no work. Furthermore, dynamic scheduling schemes can be very useful to alleviate problems of the dynamic workload variations among partitions or a poor partitioning method selection. These problems usually cannot be overcome by static scheduling schemes. Our results confirm our hypothesis.

In general, it is not always easy to determine when it is best to relocate processors in dynamic scheduling schemes. In our simulations, processors only relocate when there is no more work in their partitions. This can create problems, since a partition with a few tasks with low LVRTs is attractive to a relocating processor. Now, if many processors try to relocate at the same time they all move to the same attractive partition. This is called partition saturation and is discussed in Section **7.10.4**. Once this occurs, the large number of processors in the partition may cause scheduling irregularities. Once processors migrate to a partition they cannot leave if there is still work in the partition. This could be bad if the partition originally had a few messages with very low VRTs, but all future messages it received had very high VRTs. This would force the processors to process many of these messages with high VRTs prematurely, thus causing more backups.

Time delays between successive input messages to a task may cause many additional backups. These time delays can either be in the virtual time domain or in the real time domain. Virtual-time-delay effects are sensitive to inter-message delays between input messages in virtual time (MIT) relative to the difference in virtual time between a task's input messages and their corresponding output messages (ND). In fact, the ratio of MIT:ND is the key parameter. On the other hand, real-time-delay effects are concerned with the inter-message delays between input messages in real time (IM) in comparison to the processing time delays in real time at each task (PT). Again the ratio of $\frac{IM}{PT}$ is the key parameter here. Increasing the inter-message delays in real time (increasing IM while keeping PT fixed) or virtual time (increasing MIT while keeping ND fixed) will increase backups. Our results show that only the ratios are important, since scaling both the numerator and the denominator by the same amount does not change the timing relationship of any events in either virtual or real time. Even though these ratios are usually fixed in a given simulation, one can use the knowledge of their effects to help determine what partitioning method or scheduling scheme is likely to perform the best.

Let us consider simulations that do not have directed cycles in the flow of messages between tasks. In such simulations, a precedence relationship can be defined between any two tasks. Input nodes have the highest precedence. A task T is said to have higher precedence than task U if T is fewer links away from an input node than U on the shortest path from an input node. Tasks T and U are said to be of the same precedence if they are the same number of links from an input node on their respective shortest paths. A precedence class is a group of tasks all with the same precedence. In the network simulation there are clearly defined precedence classes that correspond to the partitions in vertical partitioning. Partitions also have precedence relations. There is said to be a precedence relationship between two partitions if each task in one partition has higher precedence than each task in the other.

We quickly learned that the most promising-looking partitioning methods did not always perform the best, and that one of the least promising-looking partitioning methods performed quite well. For example, if one were to partition a 16-input-node butterfly network (as shown in Figure 5.4) with the same number of messages into each input node, such that the messages were equally likely to have any of the output nodes as a destination, one might choose to partition the network horizontally, as in Figures 5.5 or 5.6. In this manner, one would reason that each partition would have approximately the same amount of work and that all partitions could process concurrently, since each partition has nodes in every stage through which a message has to pass to reach its destination. Also, since the different nodes in each horizontal partition depend upon each other, it is logical to group them together so as to minimize the chance that one node would get far ahead of another in virtual time.

On the other hand, one would not intuitively choose to partition the network vertically as shown in Figure 5.7. In vertical partitioning, one would reason that the input partition would be very busy at the beginning of the simulation whereas the other partitions would not be. On the other hand, at the end of the simulation the output partition would be very busy whereas the others would not be. This method also appears to be bad since none of the nodes within a partition depend upon each other. Since nodes within partitions do not communicate with or depend on each other, it does not seem necessary or valuable to prevent different nodes within the same vertical partition from getting ahead of each other in virtual time.

This analysis turns out to be a fallacy, since vertical partitioning performs many times better than horizontal, yielding fewer backups and faster processing times.

Why is the intuitive reasoning wrong? There are three reasons:

1. The synchronization effect,
2. The aggressive backup effect, and
3. The virtual-time delay effect.

The synchronization effect means that tasks within a partition tend to maintain local virtual times (LVTs) that are close to each other. This effect could either act favorably, as in vertical partitioning, or unfavorably, as in horizontal partitioning. In vertical partitioning, the partitions correspond exactly to the precedence classes of the butterfly network simulation; therefore, the synchronization effect reinforces the precedence relationships between partitions, which is good. In horizontal partitioning, the partitions contain more than one precedence class; therefore, the synchronization effect tries to force tasks of different precedence classes to have similar LVTs, which is bad, as discussed in Section **7.3**.

When a backup occurs at a task R, the set of messages Q, that were already processed are placed back on the input message queue of R. Reprocessing the messages in Q too quickly can cause additional backups. This is called the aggressive backup effect. It occurs whenever R has two sources of input messages, one from inside its own partition and another from a different partition, O. The messages in Q are reprocessed too quickly because an inappropriately high priority is given to R (within its partition) due to its low LVRT. Once this occurs, another message from O will once again cause a backup. This effect was only present in horizontal partitioning. The aggressive backup effect is discussed in more detail in Section **7.4**. This effect proved to be the major factor in determining whether lazy message cancellation improved performance in the network simulation. Lazy message cancellation improved performance only when the aggressive backup effect was present.

Virtual-time delay effects are present in horizontal partitioning, but not in vertical partitioning. Virtual-time delay effects are governed by the ratio MIT:ND. Virtual-time delays are lowered by decreasing MIT while keeping ND fixed. If there exists a precedence relationship between tasks in a simulation, then one can force the simulation system to follow this precedence relationship better by decreasing the virtual-time delays. This will make the nodes with higher precedence classes more attractive and the system will behave as if it had vertical partitions. The results show that this will reduce the number of backups in the system. Likewise, increasing the virtual-time delay will increase the number of backups. Virtual time delay effects are discussed in more detail in Section **7.5**.

Our experience suggests that it is good to form partitions such that each node is either an *input* node or a *process* node. *Input* nodes receive messages only from outside the partition, whereas the *process* nodes receive messages only from inside the partition. In other words, we discovered that it was not wise to have nodes that received input messages from both outside the partition and within the partition. This holds true for a third way to partition the network, called minimum-communications partitioning, which is shown in Figures **5.8 - 5.10**.

In vertical partitioning, all partitions that contain tasks that are N links from the input nodes have higher precedence than partitions that are N+1 links away. Like vertical partitioning, minimum-communications partitioning has precedence relationships among its partitions. In minimum-communications partitioning, the precedence relationships among partitions only exist between partitions that communicate.

In minimum-communications partitioning, each *process* node receives input from more than one task within the partition. This makes minimum-communications partitioning susceptible to partition saturation, which is when an overabundance of processors in a partition leads to an increase in the rate of backups. Since *process* nodes have two sources of input messages, a race condition may occur if both of the source nodes currently have processors that are sending messages to the same destination. If the message with the higher VRT arrives at the destination first and the destination has an empty message queue, then backup will occur if a processor starts executing the destination task before the message from the other source arrives. Thus, whenever a partition has *process* nodes with more than one input source within a partition, the partition is susceptible to partition saturation. As mentioned before, overabundance of processors can easily occur in any dynamic scheduling scheme and can cause partition saturation.

High virtual-time delays (ND=0) can sometimes increase the effects of partition saturation by making all the nodes equally attractive. Therefore, the extra processors may choose to execute nodes with lower precedence than other available nodes causing more backups to occur. In this case the virtual-time delay and partition saturation effects are compounded and the results are worse than if only one effect were present. This was discussed in Section **7.10.4**.

As was the case in partitioning methods, scheduling schemes also were not always as good as expected. One of the scheduling schemes that was used in our experiments was the continuous dynamic repartitioning scheme. In this scheme, every time a processor finished processing a task, the processor relocated itself to the partition with the lowest LVRT and processed the task with the lowest LVRT currently on the partition's task queue. In the other dynamic schemes, a processor would only relocate after processing a task if there was no more work in its current partition. We thought the continuous scheme would perform well because:

1. It appeared to exploit partitioning, thus alleviating the problem of task-queue contention in the non-partitioning case.
2. It appeared to assure that the next task processed had the lowest LVRT, unlike the other dynamic schemes.

However, the continuous scheme did not outperform the other schemes. In fact the continuous scheme had many more backups than other dynamic scheduling schemes when a good partitioning method was employed.

Where did we go wrong again? The continuous scheme did not respect partition boundaries very well. Therefore, for the butterfly network simulations, the performance of the continuous scheme is mainly characterized by the ratio of MIT:ND, making the continuous scheme very susceptible to virtual-time delays. As the MIT increased for fixed ND, the number of backups and processing times for the continuous scheme skyrocketed, regardless of the partitioning method used.

Meanwhile, the dynamic schemes that obeyed partition boundaries were largely unaffected by large virtual-time delays in vertical partitioning, and were only slightly affected in minimum-communications partitioning. Good partitioning tends to shield the scheduling schemes from virtual-time-delay effects by forcing explicit scheduling priorities. For example, in vertical partitioning, processors tend to relocate to the input partitions at the beginning of the simulation because only the input partitions have work at that moment. Once the processors migrate there, they tend to stay there until all of the input messages are processed. This tends to be optimal in preventing backups.

One important factor is the average message queue length of tasks in the system. Longer real-time delays for input messages entering the simulation shorten the message queue lengths, because they delay messages from entering the system. Shorter message queue lengths increase the chance of backups as shown in Section **7.7**. On the other hand, when backup occurs, many messages must be reprocessed. This adds already processed messages onto a task's input queue, while it adds additional anti-messages into the system to cancel processed messages. Overall, this will add more messages into the system, and thus increase the average message queue length. Therefore, there is a relationship between average message queue length and backups which eventually causes a message queue equilibrium that differs for each partitioning method. For vertical partitioning, the average message queue length at equilibrium is much higher than for horizontal partitioning. This coincides with fewer backups in the vertical case.

The Time Warp system has many attributes that are similar to those found in the Network Paradigm system. In the Network Paradigm system there were two important points that have counterparts in the Time Warp system. These two important points are safety and feedback.

For the Network Paradigm system, safety can be split into two cases:

1. The safe case - a task is in this case when all of its input queues have at least one message. As long as the safe case exists, it is safe for the task to execute at least one message off its input queues.
2. The unsafe case - a task is in this case when at least one of its input queues is empty. As long as the unsafe case exists, it is unsafe for the task to execute any of its messages, since a message could later arrive on an empty input queue with a VRT less than the message just processed. In this case the task must wait until

the safe case occurs.

Feedback is also an important point in the Network Paradigm. Feedback occurs when the network topology has directed cycles. This can cause deadlock, and may require deadlock detection and resolution schemes.

In the Time Warp case each task has only one input message queue. For the Time Warp system, safety and feedback have analogies. Safety can be divided into two cases:

1. The semi-safe case - a task that has a very long input message queue is in the semi-safe case. In this case, if the message with the lowest VRT is processed at this task, then backup is very unlikely, because the probability that a new message could arrive with a lower LVRT than the message just processed is small. Nevertheless, backups can only be avoided entirely if we process messages only at the task with the lowest LVRT in the entire system (globally).
2. The unsafe case - a task that has a very short input message queue is in the unsafe case. In the extreme case, the input message queue is empty, which corresponds exactly to case 2 of the Network Paradigm system. In the Time Warp case, processing messages of a task in the unsafe case will likely cause backup.

Feedback is also important in the Time Warp case. Here, feedback increases the real-time delay between successive messages which in turn decreases the message queue length. The feedback case causes many backups in the Time Warp system.

In the Time Warp system, simulations with feedback generally have a smaller number of input messages into the system. The fewer messages in the system cause shorter message queue lengths, which in turn cause more backups.

8.3. Additional Questions

There are two major issues that were not investigated in depth in this thesis. They are:

1. Task Granularity and
2. Feedback

Although granularity was not investigated in this thesis, it is a very important issue. Granularity is the size of a task. We handle granularity by making each task a logical element of the simulation. A related thesis by Arnold[1] discusses a different approach. The logical elements of his simulation are called atomic units. Each of his tasks (he calls these partitions) contain any number of atomic units. When backup occurs, the state of all the atomic units in a task must be restored. The number of tasks in a simulation equals the number of physical processors. The research attempted to give each task approximately the same number of atomic units. The research does not discuss how to assign atomic units to tasks.

The major drawback with his approach is that even if only a few atomic units of a task need to back up, all of them are backed up. This does not occur in our system because our tasks are equivalent to his atomic units. Therefore our backups are localized. Although we have chosen our tasks to be small atomic units, there are no restrictions in our system requiring tasks to be chosen in this manner.

Even though Arnold's thesis experimented with varying task sizes, his experiments were limited to six processors and static scheduling. Therefore, task granularity is still a major area where additional work could be done.

The other major area where additional work could be done is simulations with feedback. What is the best partitioning method to use in feedback systems? How much do virtual-time delays affect feedback? Can additional null messages reduce backups in feedback systems?

In the Network Paradigm system, a partial solution to the unsafe case is to send null messages between linked tasks that have not communicated for a specified period of real time. Similarly, to increase the message queue length in the Time Warp system, one might envision a plan to introduce many null messages into the system to help keep the tasks synchronized. This might be especially useful in feedback systems where the message queue lengths tend to be much shorter than in the non-feedback systems. The question is how much extra processing time is required to process all of the additional null messages? Is the amount of processing time devoted to processing these null messages a small enough percentage of the total processing time to make it worth doing?

During this thesis research, it became clear that running larger examples was more informative and gave more consistent data (average message arrival time plotted against either processing time or number of backups). In 16-input-node network simulations, versus 8-input-node network simulations, the effects of varying parameters became more pronounced. Unfortunately, Concert, the multiprocessor used for our work, proved to be very prone to crashes in the larger simulation regime. Future research needs to be directed towards these larger simulations, those having more nodes, more partitions, and on reliable machines having more than 32 processors. It will become most important to determine how our system performs on these larger scales.

IX. APPENDIX

This appendix illustrates the six different random partitions used on a 16-input-node butterfly network. These were originally introduced in Section **5.5.2**. They were:

1. R_{8a} with 8 partitions is shown in Figure **9.1**.
2. R_{8b} with 8 partitions is shown in Figure **9.2**.
3. R_{8c} with 8 partitions is shown in Figure **9.3**.
4. R_{16a} with 16 partitions is shown in Figure **9.4**.
5. R_{16b} with 16 partitions is shown in Figure **9.5**.
6. R_{16c} with 16 partitions is shown in Figure **9.6**.

Partitions belonging to R_{8a}, R_{8b} and R_{8c} were made to contain 12 tasks each, whereas partitions belonging to R_{16a}, R_{16b} and R_{16c} had 6 tasks each chosen randomly using a uniform random variable. These tasks were either nodes, drivers or probes.

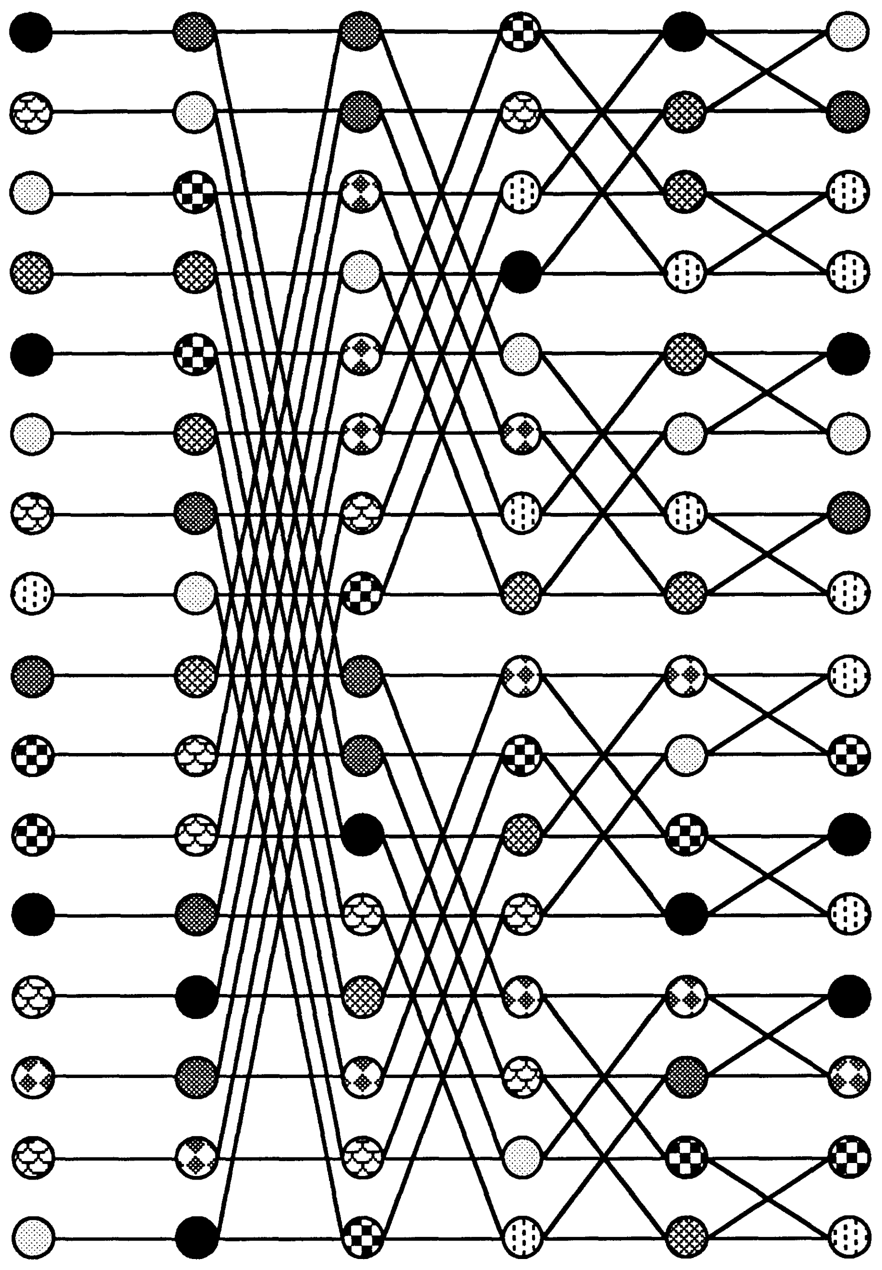


Figure 9.1 Random Partitioning I with 8 Partitions R_{8a}

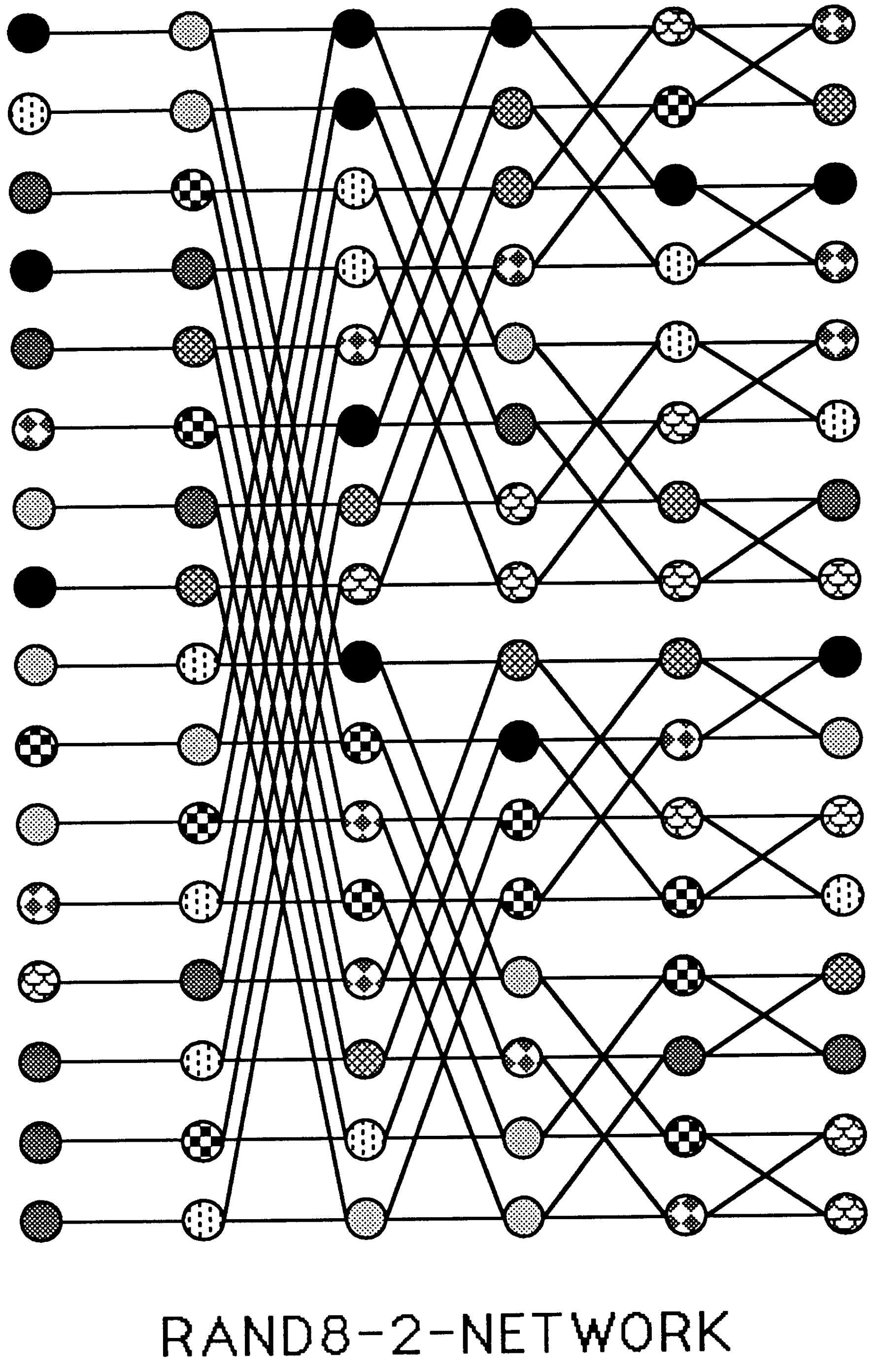


Figure 9.2 Random Partitioning II with 8 Partitions R_{8b}

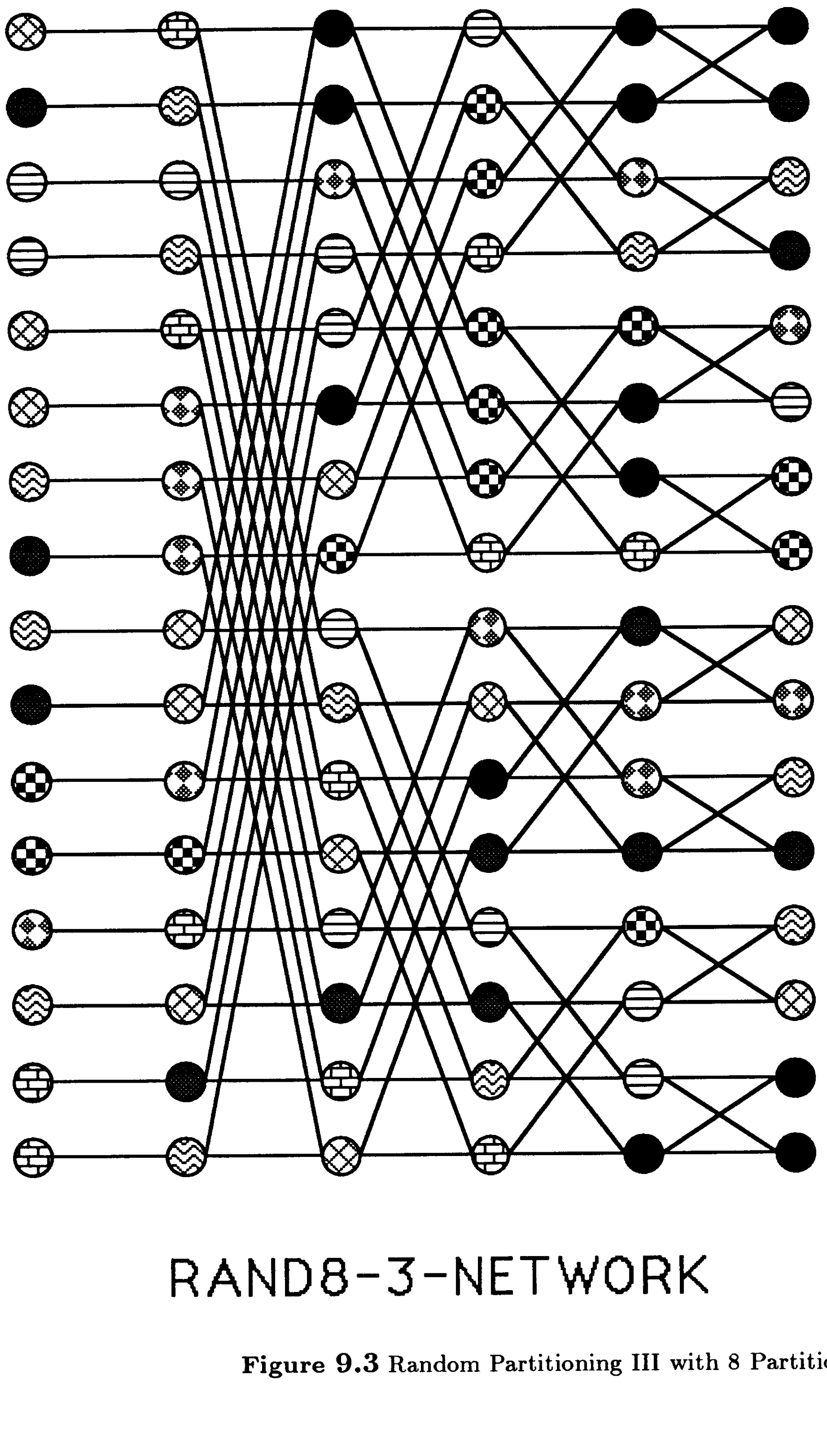


Figure 9.3 Random Partitioning III with 8 Partitions R_{8c}

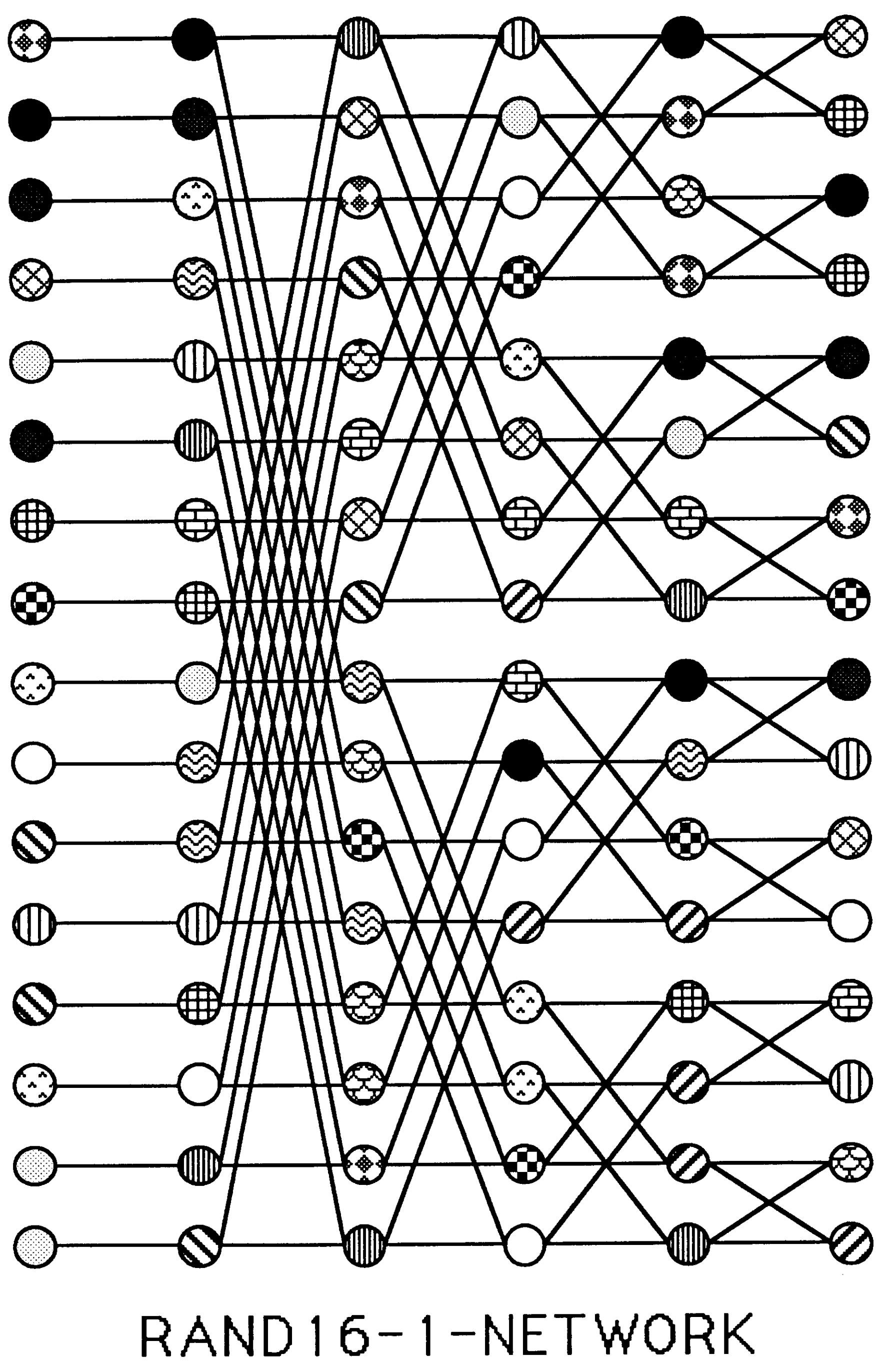


Figure 9.4 Random Partitioning I with 16 Partitions R_{16a}

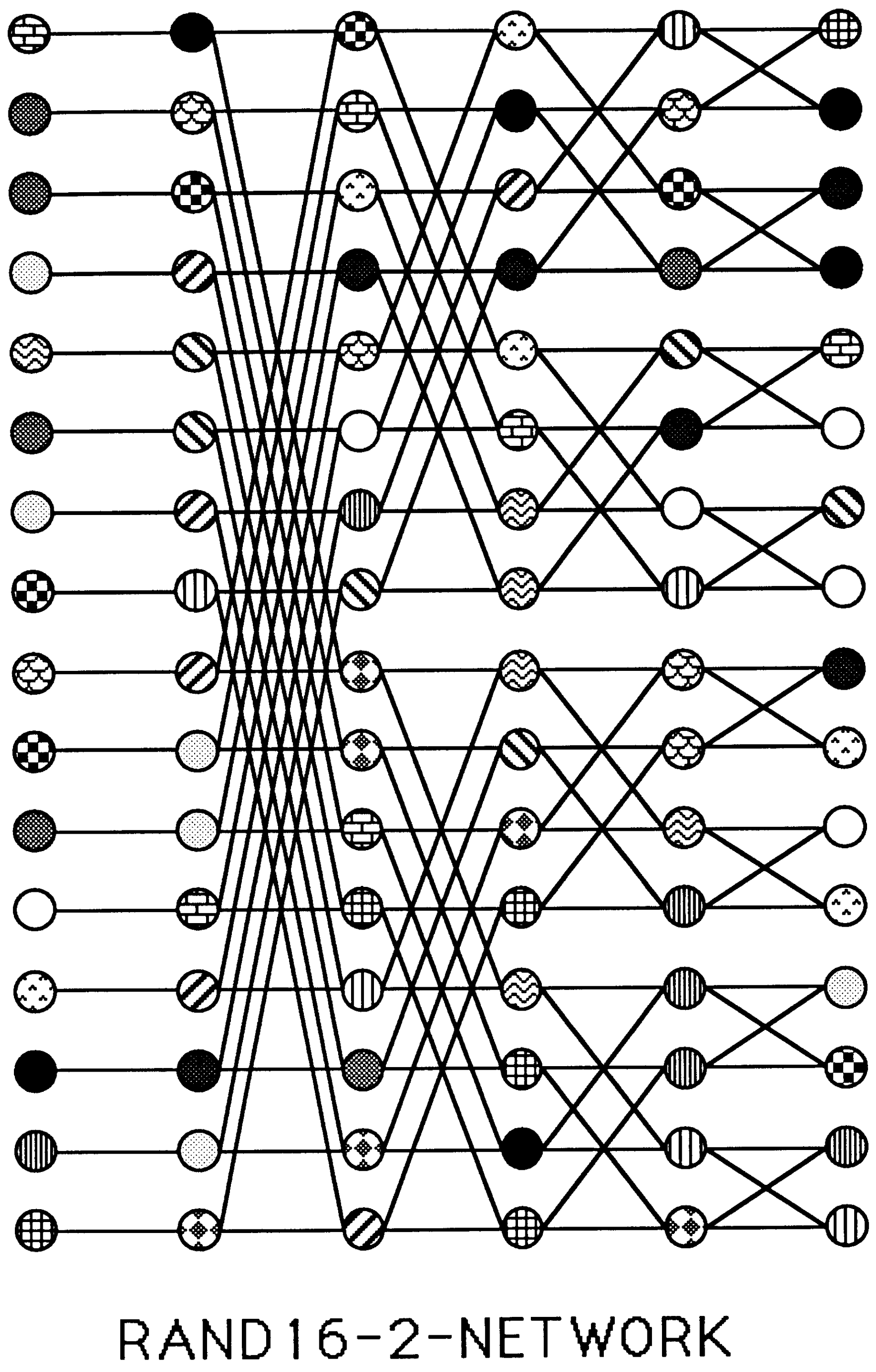


Figure 9.5 Random Partitioning II with 16 Partitions R_{16b}

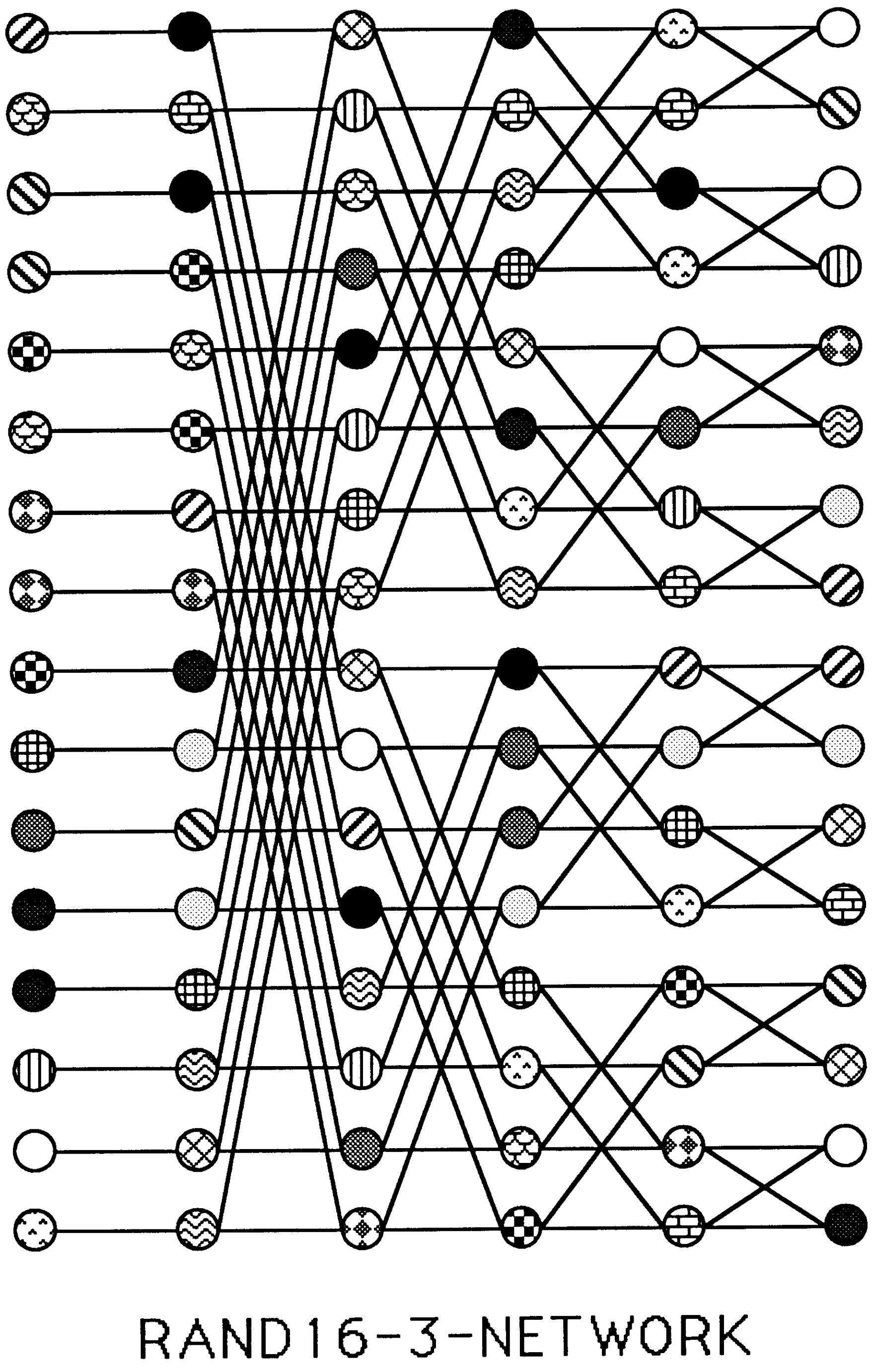


Figure 9.6 Random Partitioning III with 16 Partitions R_{16c}

References

[1] Arnold, J., "Parallel Simulation of Digital LSI Circuits", Tech. Rep. TR-333, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Mass., Feb. 1985.

[2] Anderson, T., "The Design of a Multiprocessor Development System", Tech. Rep. TR-279, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Mass., Sept. 1982.

[3] Berry, O., and Jefferson, D. R., "Critical path analysis of distributed simulation", 1985 Society for Computer Simulation Multiconference (San Diego, Calif., Jan. 1985).

[4] Bryant, R. E., "Simulation of packet communication architecture computer systems", M.S. dissertation, M.I.T., Nov. 1977.

[5] Chandy, K. M., and Misra, J., "Asynchronous distributed simulation via a sequence of parallel computations", Communication ACM 24, 4 (Apr. 1981), 198-206.

[6] Chandy, K. M., and Misra, J., "Distributed simulation: A case study in design and verification of distributed programs", IEEE Transaction on Software Engineering, SE-5, 5 (Sept. 1979), 440-452.

[7] Halstead, R. H., Jr., "Multilisp: A Language for Concurrent Symbolic Computation", ACM Transactions on Programming Languages and Systems 7, 4 (October 1985), 501-538.

[8] Jefferson, D. R., and Sowizral, H. A., "Fast concurrect simulation using the Time

Warp mechanism, partI: Local control", Rand Note N-1906AF, The Rand corp., Santa Monica, Calif., Dec. 1982.

[9] Jefferson, D. R., and Sowizral, H. A., "Fast concurrent simulation using the Time Warp mechanism", Proceedings of the SCS Distributed Simulation Conference (San Diego, Calif., Jan. 1985).

[10] Jefferson, D. R., ET AL, "Implementation of Time Warp on the Caltech Hypercube", 1985 Society for Computer Simulation Multiconference (San Diego, Calif., Jan. 1985).

[11] Jefferson, D. R., and Witowski, A., "An approach to Performance analysis of timestamp-driven synchronization mechanisms", Proceedings of the 3rd ACM Annual Symposium on Principles of Distributed Computing, (Vancouver, B.C., Canada, Aug. 1984), ACM, New York.

[12] Jefferson, D. R., and Motro, A., "The Time Warp mechanism for database concurrency control", U.S.C. Tech. Rep., Dept. of Computer Science, Univ. of Southern California, Los Asngeles, June 1983.

[13] D.R. Jefferson, "Virtual Time", ACM Transaction on Programming Languages and Systems 7, 3 (July 1985), 404-425.

[14] Lamport, L., "Time, clocks, and the ordering of events in a distibuted system", Communication ACM 21, 7 (July 1978), 558-565.

[15] Lavenberg, S., Muntz, R., and Samadi, B., "performance analysis of a rollback method for distributed simulation", Dept. of Computer Science, U.C.L.A., 1982.

[16] Peacock, J. K., Wong, J. W., and Manning, E. G., "A distributed approach to queueing network simulation", 1979 Winter Simulation Conference, IEEE, New York, 1979, 399-406.

[17] Peacock, J. K., Manning, E. G., and Wong, J. W., "Synchronization of distributed simulation using broadcast algorithms", Computer Networks 4, 1 (Feb. 1980), 3-10.

[18] Peacock, J. K., Wong, J. W., and Manning, E. G., "Distributed simulation using a network of processors", Computer Networks 3, 1 (Feb 1979), 44-56.

[19] Samadi, B., "Distributed simulation: Performance and analysis. Ph. D. dissertation, Dept. of Computer Science, UCLA, Los Angeles, 1985.

[20] Schneider, F. B., "Synchronization in distributed programs", ACM Transactions on Programming Language Systems 4, 2 (Apr. 1982), 179-195.

[21] Sowizral, H. A., "The Time Warp simulation system and its performance", 1985 Society for computer Simulation Multiconferece (San Diego, Calif., Jan. 1985).

[22] Wyatt, D., Sheppard, S., and Young, R., "An experiment in microprocessor-based distributed digital simulation", Proceedings of the 1983 Winter Simulation Conference (Arlington, Va., Dec. 1983), S. Roberts, J. Banks, and B. Schmeiser, Eds.