RICE UNIVERSITY

A Loop Structured Technique for the Control of a Multiprocessing System.

by

William Lamar Bain, Jr.

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

Thesis Director's Signature:

Houston, Texas

May, 1975
ABSTRACT

A Loop Structured Technique for the Control of a Multiprocessing System.

by

William Lamar Bain, Jr.

A method for routing control information to and from the processors in a multiprocessing system is presented and analyzed. The technique requires the construction of two sets of data registers connected in the form of closed loops. Every processor is interfaced to one register on each loop through which data is propagated in a parallel manner and at a constant rate (as controlled by a system clock). A system controller introduces data words called "packets" onto the "input" loop for distribution to available processors. The packets contain information sufficient for the initiation of concurrent tasks. The processors input these packets, execute tasks as directed, and generate similar packets which signify task completion. The latter packets are deposited on the "output" loop by the processors for return to the system controller.

The analysis reveals that under appropriate restrictions (namely, a large supply of tasks, a fixed task execution time, and the efficient handling of packets by the system controller), the scheme makes effective use of processing resources. The throughput of tasks increases linearly with the addition of processors until the number of processors, n, is equal to the ratio of task execution time to system clock time, p, after which it remains fixed as more processors are
added. The overhead required to route control information through the system decreases as $1/n$ until $n$ equals $p$, whereupon it increases linearly for larger $n$.

A simulator, used to verify the analysis and extend same to more general systems, is also described. It shows that as restrictions are lifted, the loop structured technique continues to be a viable scheme for the control of a multiprocessor, especially in applications in which its simplicity and modularity are advantageous.
ACKNOWLEDGEMENTS

I would like to gratefully acknowledge the support and direction given to this work by Dr. J. Robert Jump. His intelligence and professionalism are equaled by few people. I also express my appreciation to Dr. Edward Page, Dr. Edwin Feustel, and Dr. Sigsby Rusk for serving on my thesis committee and providing their ideas, criticisms, and suggestions. Of my fellow graduate students, Sudhir Ahuja deserves special mention for lending his talents to problems encountered in this research. The typist, Ms. Pat Angel, struggled long hours to decode the manuscript and has my sincere thanks. Finally, I am indebted to my good friends Rick White, Bob Arnold, and Ann Howland, for making my graduate life a pleasure.

This work was supported by N.S.F. Grants GJ 750, DCR 7414283, and GJ 36471.
Chapter I
INTRODUCTION

As the speed of computing hardware reaches theoretical limits [4], alternative methods of increasing computing power* must be considered. For this reason, multiple processors have been incorporated into a single system (herein called a "multiprocessor"). The effective utilization of the processing resources in a restricted class of such systems, namely those in which the power of an individual processor is relatively limited, is the aim of this thesis.

Multiprocessing originated about 1960 with the Univac Larc computer. A second processor dedicated to input-output processing was added to a single processor system in order to free the central processor for useful computation, thus increasing overall computing power. Another form of multiprocessing, in which multiple, identical processors performed the same computations and compared results, became popular in applications demanding high reliability (e.g., on board spacecraft). Computing power was not enhanced, but processing continued if a processor failed. A logical hybrid of the two above techniques is the so-called "symmetrical" multiprocessor which consists of several identical processors normally performing different tasks concurrently. This system, if properly designed, has much greater computing power than a single processor system. Reliability is enhanced by having more than one processor capable of executing all tasks; should a processor fail, tasks can be rerouted to the

*Computing "power" is to be interpreted here as the number of useful operations performed in a given unit of time.
remaining processors. Finally, the system lends itself to modular
design, thus being easily expanded without major changes in the hard-
ware or software design philosophies.

It is clear that the effectiveness of a multiprocessor depends
on the care placed in its design so that it meets the important
criteria of

a) efficient execution of tasks,

b) reliable performance and "graceful" degradation
   in case of processor failure, and

c) expandability with minimal change to the design.

Research into multiprocessor design to meet these criteria can
be divided into three categories [1]:

(1) Data flow structures: The data paths in a multiprocessor
   reflect a tradeoff between desired performance (via flexibility of
data flow between processors, memories, and devices) and hardware
costs. Davis, Zucker, et al. [2] compare several schemes considered
in the design of the Burroughs modular multi-interpreter. Among
those are the single bus, multiple bus, matrix or cross-bar switch,
and multiport schemes.

(2) Formulation of tasks for concurrent processing: Research
   in this crucial area has proceeded along several lines. Program-
mimg languages (e.g., Pascal) and programming language constructs
have been devised to allow the user to explicitly express concur-
rency in his program (e.g., FORK-JOIN of Dennis) and to synchronize
program segments capable of concurrent execution (e.g., P,V of
Dijkstra). Algorithms for the automatic detection of parallelism
have been devised, placing concurrency on the expression and statement level as well.

(3) Processor control and synchronization: Multiprocessors present a significant challenge in the design of a control mechanism that makes effective use of the system's computing resources (i.e., the processors). Control can be centralized in a processor dedicated to the allocation of system resources (Univac 1108) in which case the processors are allocated tasks when necessary by this "master" processor; alternatively control can be distributed among all the processors in the system (Burroughs D825), wherein the processors actively schedule new tasks for themselves as required [2]. In both of the above cases, a software algorithm is used to determine which tasks are ready for execution and to which processors they are to be allocated. The relatively long overhead required to implement a sophisticated control algorithm is offset by the tasks's duration in execution by a given processor. To implement multiprocessing on a lower level (e.g., single machine instructions) requires a simpler strategy often effected directly in the hardware (e.g., the IBM 360/91 Common Data Bus as described by R. Tomasulo [7]).

With the development of the microprocessor, an inexpensive processor of limited computational capability constructed with a small set of LSI integrated circuits, multiprocessor design takes on a new complexion. Whereas a microprocessor is limited in power, several such devices acting in concert could yield a very powerful system indeed. Multiprocessor research efforts (as enumerated
above) must then be reoriented from the "macro" processor mentality of the past to the dissimilar characteristics of the new devices. In particular, the complex control strategies found in the large multiprocessor systems must be abandoned in favor of simpler but still effective techniques. This thesis describes and analyses one such technique, the use of loop structured data paths for the distribution and collection of control information.

The thesis will describe in detail in the next chapter a method for routing control information to and from a coordinated system of processors. Chapter 3 will provide a deterministic analysis on a restricted (but viable) version of this structure suitable for such analysis. Chapter 4 will provide a description of a PL/I simulation program used to verify the analysis of the restricted structure and predict the performance of the structure in its full generality. Simulation results will be presented here. A discussion of possible extensions to this work will follow in Chapter 5. It should be noted that while the structure herein described is motivated by the potential seen in microprocessor technology, its relevance is not confined to it. Any system design in which the routing of control information to multiple processors is of concern could consider its application.
Chapter 2
THE DESIGN

The control mechanism for a multiprocessor system can be defined as that computing resource capable of performing the following functions:

A. division of the computing requirements of the system users into a set of "tasks" suitable for processor execution,
B. distribution of the tasks to the processors, and
C. disposition of processing results so that new tasks may be formulated and distributed.

Note that these functions must be performed on a continuous basis. The above stated duties of the control mechanisms suggest two criteria by which its effectiveness can be measured:

1. System throughput: How many tasks can be performed by the system per unit time?
2. System overhead: How much of the computing resources (i.e., time and storage) are required to support the control mechanism?

These form the fundamental standard by which the multiprocessor control scheme to be presented will be evaluated.

The literature contains many examples of multiprocessor control schemes (in particular, see Enslow [4]). They may conveniently be divided into two categories as previously stated, namely centralized and distributed strategies. The advantages and disadvantages of each motivate the control mechanism to be discussed. The centralized approach uses a dedicated "central" processor to carry
out the three major functions (see figure 2-1). Sophisticated, efficient algorithms for task formulation and processor allocation are possible; the bottleneck occurs in the communication with the (peripheral) processors. Since there is only one (in general) central processor, the peripheral processors can be activated only one at a time (function B). Further, processors must interrupt the central processor to signal completion or, alternatively, the central processor must scan the processors to determine their status (function C). There is, then, a large amount of overhead associated with the centralization of control in terms of both control path hardware and central processor execution time.

The distributed approach contains no central processor; each processor in the system schedules its own tasks as required (see figure 2-2). The communication problem of the centralized scheme is gone as there is no single processor directing activities. However, a method of coordinating the now autonomous processors must be devised. A list of executable tasks storable in common memory could be constructed and updated by a "system" task that runs periodically (function A). The processors obtain new tasks (function B) and signal completion (function C) by accessing the list (and other system tables as necessary). Access must be limited to one processor at a time, of course; hence a processor interlock mechanism must be constructed, requiring additional complexity in both hardware and software. (See Davis, et al. [2] for a detailed development of this scheme and Levy [6] for an alternative scheme in which processors activate other processors.)
FIGURE 2-1: A CENTRALIZED MULTIPROCESSING SYSTEM

FIGURE 2-2: A DISTRIBUTED MULTIPROCESSING SYSTEM
Although sophisticated in their performance, the distinctly
centralized or distributed systems are not feasible in applications
where processor power is not commensurate to their complexity. The
loop structured system provides a simple yet effective alternative.
It is assumed that all processors can execute all tasks as formulated (there are no special purpose processors). Further, assume
that no interprocessor communication is required; clearly tasks
can be delimited so that this is possible. Thus, the processors
are "anonymous" with respect to the tasks; any processor can execute any task. The loop structured system is oriented around two
one-way loops, an "input loop" and an "output loop". Each loop
consists of a set of registers connected in a circular manner and
tied to a common (continuously running) system clock so that each
clock pulse causes a parallel data transfer from each register to
its successor. There is a register on each loop for every processor in the system, to which the processor is interfaced. Tasks
are formulated as data words called "input packets" and placed
sequentially into a specific input loop register (denote it as
register 1) by a "supply" device. A packet contains information
sufficient to direct a processor to begin execution on a given task.
Each processor examines its respective input registers for a
packet (a one bit tag in the packet indicates that the register
contains data versus a blank), removes the first available packet
(by changing the tag), and begins execution on the task. Input
packets circulate in their loop until some processor removes them.
Upon completion of a task, the processor constructs an "output
packet" which it places on the output loop when its output register contains a blank. An "acceptor" device interfaced to a specific output loop register (the predecessor to register 1) removes output packets and updates system tables as required. (See figure 2-3.)

The supply and acceptor devices, together called the "system controller", are largely undefined and can be designed around the constraints of the system. (They could be implemented as a single processor if necessary.) They can handle all of the task formulation and disposition functions of the system or, alternatively, let part of their work be done by the processors (as system tasks) and serve in a minimal capacity to update system tables and handle packet traffic onto and off of the loops. Their importance lies in the fact that the issuing of tasks and the updating of system information is handled by a single processor, and thus processor interlocks are unnecessary. The supply is not responsible for the distribution of tasks to the processors (as is a central processor in the centralized system); this duty is properly handled by the input loop and processor demand.

The two loops have several advantages. The input loop, as stated, provides an automatic mechanism for the distribution of tasks to the processors. The system clock can hopefully drive packets around the loops very rapidly relative to task execution times. Thus, processors and the acceptor need wait relatively short amounts of time for the arrival of a packet at their respective latches. The loops form a convenient basis for the modu-
THE LOOP STRUCTURED CONTROL SCHEME

S = supply device
I.R. = input register
P = processor
O.R. = output register
A = acceptor

FIGURE 2-3
larization of the system, also. A processor and its two loop
registers constitute a convenient module, allowing facile expan-
sion or reconfiguration in case of processor failure. The loops
serve to buffer the relative speeds of the supply, processors, and
acceptor during varying system loads (where the "load" is the
number of tasks being generated per unit time). Note that all
three run asynchronously although they must be synchronized at the
loop interfaces. If the entire system is driven by a single clock,
then this is a reasonable requirement. Sample designs of such
interfaces (assuming synchronized signaling) are found in figure
2-4; they indicate that the implementation can be simple and
economical.

The precise contents of the packets greatly depends upon the
memory management schemes employed. The input packets presumably
could contain fields for task identification, the operation to be
performed, and the accessing of relevant data. Output packets
could contain fields for task identification, accessing of results
(data), and the possible generation of new tasks. The relative
amounts of memory dedicated to each processor or common to the
system would influence the complexity (and, thus, size) of these
packets. It is, of course, assumed that separate data paths to
common memory would be devised (e.g., a bus); the loops are only
for the transmission of control information. Davis, et al. [2] sug-
gest that processors be dynamically microprogrammable, in which case
the input packet would identify the appropriate microprogram to be
employed for the execution of this task. Processors could then
be used for general purpose computing, input/output controlling, etc. Chapter 5 will expand upon these ideas.

Detailed system characteristics follow.

Assume there exist \( n \) processors, labelled \( 1, \ldots, n \).

1. Each loop has \( n \) registers, labelled \( 1, \ldots, n \).

2. Processor \( i \) may remove data only from input loop register \( i \); it may place data only on output loop register \( i \), \( i = 1, \ldots, n \).

3. All data transfers take place at the arrival of clock pulses from the system clock. The system clock generates pulses at a constant rate. Denote these pulse times as \( 0, 1, 2, 3, \ldots \) or \( t \in \mathbb{Z}^+ \).

4. Processor \( i \) may remove a packet from input register \( i \) at time \( t \) if register \( i \) captures a packet at time \( t-1 \). A blank is then placed in register \( (i \mod n) + 1 \) at time \( t \), \( i = 1, \ldots, n \).

5. Processor \( i \) may place a packet in output register \( i \) at time \( t \) if register \( i-1 \) captures a blank at time \( t-1 \), \( i = 2, \ldots, n \).

6. Processor 1 may place a packet in output register 1 at time \( t \) if register \( n \) captures a blank at time \( t-1 \) or if the acceptor signals its intention to take a packet from register \( n \) at time \( t \) (via a special signal path).

7. The supply may place a packet in input register 1 at time \( t \) if register \( n \) captures a blank at time \( t-1 \) or if proces-
sor n signals the supply of its intention to take a packet from register n at time t (via a special signal path).

8. The acceptor may remove a packet from output register n at time t if register n captures a packet at time t-1.

9. If none of the above transfers takes place at time t, then the contents of input (output) register i will be placed in input (output) register \((i \mod n) + 1\) at time t.

10. A processor may take a packet from its input register at the same clock pulse in which it deposits an output packet in its output register.
Chapter 3
THE ANALYSIS

It is important to determine the behavior of the loop structured system, particularly with respect to the previously defined criteria of system throughput and overhead. In this chapter, a deterministic analysis of a restricted version of the system is presented. The simulator (described in the following chapter) will be used to verify the results produced and elucidate complex behavior beyond the scope of the analysis.

The following assumptions are motivated to simplify the intricate interactions between the components of the system:

1) There is a continual supply of tasks to be performed; the supply device is always busy.
2) The supply device is capable of constructing an input packet in less than one clock period and is ready to supply a packet when the first clock pulse is generated.
3) The acceptor device is capable of disposing of output packets in less than one clock period.
4) All tasks require exactly the same amount of processing time within the processors. Denote this time as $T_p$.
5) There is no waiting for input/output of data to common memory; any such operations take place without delays.
6) All processors are initially idle.

The first two assumptions together imply that whenever the supply device has the opportunity to deposit an input packet in input register 1 (according to previously stated constraints), it will do
so. It must be possible to predict when packets will enter the system in order to carry out a deterministic analysis. These two assumptions accomplish this (clearly, alternatives are possible, e.g., a pipelined supply device, in which case assumption 2 would not be necessary). Assumption three effectively negates the necessity for an output loop; the loop degenerates into a multiple input/one output queue since no packets recirculate. This assumption prevents output loop "conflicts" (a conflict occurs when a processor wishes to deposit an output packet but is prevented from doing so by the presence of a packet in its output register) caused by recirculating packets. It will be shown that no other output loop conflicts exist for the restricted system, and hence the output loop will not disrupt the behavior of the processors. Although a severe restriction with tasks of widely varying execution times, assumption 4 synchronizes the processors in a very desirable manner, as will be seen. Assumption 5 removes data flow problems and memory management decisions from this analysis. Whereas they are critically important, such issues are not considered here (with one exception—a single bus scheme requiring no bus arbitration will be presented). Assumption 6 serves only to simplify the analysis.

Let the system contain \( n \) processors and let \( \rho = \frac{T_p}{T_c} \) where \( T_c \) is the period of the system clock. Restrict \( n, T_p, T_c \) and \( \rho \) to the set of positive integers \( 0,1,2,\ldots \) which we denote as \( Z^+ \). Let the system clock generate a pulse at every time, \( t \), such that \( t \in Z^+ \). Then we have:

**Lemma 1**: Processor \( i \) initiates execution initially at time \( 2i-1 \)
for \( i = 1, \ldots, \min(p,n) \).

(Detailed proofs to all lemmas and theorems may be found in the appendix.) The proof shows (by induction on \( n \), the number of processors in the system) that the first packet arrives at processor \( i \)'s input register at time \( 2i-2 \). Thus, processor \( i \) fires initially at the time stated for \( i = 1, \ldots, \min(p,n) \). Consider a system with three processors as in figure 3-1. Processors 1, 2, and 3 initiate at times 1, 3, and 5, respectively, since \( p = n = 3 \). Careful observation of succeeding processor initiation times in this example suggest the following lemma:

**Lemma 2:** Processor \( i \) will initiate execution only at times \( 2i-1+j \), where \( i = 1, \ldots, \min(p,n) \) and \( j \in \mathbb{Z}^+ \). That is, processor \( i \), once started, is always busy.

The proof shows that, for every processor \( i \), there is a packet in its input register and the processor is free to begin execution at times \( 2i-1 + j \rho \) and only these times. This is accomplished by inducting on \( j \), the number of times each processor has been activated, and on \( n \), the number of processors in the system (using Lemma 1). The proof shows that for every processor \( i \) (where \( i = 1, \ldots, \min(m,n) \)), there is a packet in its input register and it is ready to begin execution at the times stated. This requires that no processor be delayed by output loop conflicts, and this is shown to be the case. Thus, the initial regimen of processor starting times is maintained over the duration of the system at intervals of \( \rho \) clock periods (due to the common processor execution time of \( \rho \)). Lemma 2 indicates that the input loop distributes
KEY TO SYSTEM BEHAVIOR
GRAPHS

TIME

SYSTEM

\( t \)

\( o_1 \) \( o_2 \) \( o_3 \)

\( p_1 \) \( p_2 \) \( p_3 \)

\( i_1 \) \( i_2 \) \( i_3 \)

\( t = \) time of system snapshot

\( i_1 = \) task in input register 1

\( i_2 = \) task in input register 2

\( i_3 = \) task in input register 3

\( p_1 = \) task in processor 1

\( p_2 = \) task in processor 2

\( p_3 = \) task in processor 3

\( o_1 = \) task in output register 1

\( o_2 = \) task in output register 2

\( o_3 = \) task in output register 3
SYSTEM BEHAVIOR: n=3, p=3

<table>
<thead>
<tr>
<th>TIME</th>
<th>SYSTEM</th>
<th>TIME</th>
<th>SYSTEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>- - -</td>
<td>6</td>
<td>- 2 1</td>
</tr>
<tr>
<td></td>
<td>- - -</td>
<td></td>
<td>4 5 3</td>
</tr>
<tr>
<td></td>
<td>1 - -</td>
<td></td>
<td>7 6 -</td>
</tr>
<tr>
<td>1</td>
<td>- - -</td>
<td>7</td>
<td>4 - 2</td>
</tr>
<tr>
<td></td>
<td>1 - -</td>
<td></td>
<td>7 5 3</td>
</tr>
<tr>
<td></td>
<td>2 - -</td>
<td></td>
<td>8 - 6</td>
</tr>
<tr>
<td>2</td>
<td>- - -</td>
<td>8</td>
<td>- 4 3</td>
</tr>
<tr>
<td></td>
<td>1 - -</td>
<td></td>
<td>7 5 6</td>
</tr>
<tr>
<td></td>
<td>3 2 -</td>
<td></td>
<td>9 8 -</td>
</tr>
<tr>
<td>3</td>
<td>- - -</td>
<td>9</td>
<td>- 5 4</td>
</tr>
<tr>
<td></td>
<td>1 2 -</td>
<td></td>
<td>7 8 6</td>
</tr>
<tr>
<td></td>
<td>4 3 -</td>
<td></td>
<td>10 9 -</td>
</tr>
<tr>
<td>4</td>
<td>1 - -</td>
<td>10</td>
<td>7 - 5</td>
</tr>
<tr>
<td></td>
<td>4 2 -</td>
<td></td>
<td>10 8 6</td>
</tr>
<tr>
<td></td>
<td>5 - 3</td>
<td></td>
<td>11 - 9</td>
</tr>
</tbody>
</table>

5    | - 1 -  |
|      | 4 2 3  |
| 6    | 5 -    |

FIGURE 3-1
tasks very efficiently; the first \( \rho \) processors are always busy. Systems with only \( \rho \) processors have optimal use made of their processor resources.

For a packet supplied at arbitrary time, \( t_s \), it is possible to determine the precise time that it will be taken from the input loop and which processor will take it. The two cases \( \rho < n \) and \( \rho > n \) will be considered separately.

**Theorem 1:** Assume \( \rho < n \). Then if an input packet is supplied at time \( t_s \), processor \( R[t_s/\rho] + 1 \) will take it from the input loop at time \( t_s + R[t_s/\rho] + 1 \).

The proof of Theorem 1 is as follows. Given the supply time, \( t_s \), the input register containing the supplied packet is known for all subsequent times. Since all processor initiation times are known (Lemma 2), the first processor \( i \) such that the packet arrives in processor \( i \)'s input register at one time unit before processor \( i \) begins execution can be determined. Thus, both the processor which takes the packet and its initiation time can be found. In figure 3-1, packet 6 is supplied at time 5 and is taken by processor \( 3 = R[5/3] + 1 \) at time \( 8 = 5 + R[5/3] + 1 \).

There are two important consequences to Theorem 1. First, since \( R[t_s/\rho] \leq \rho \) it is clear that input packets never recirculate in the case that \( \rho < n \) (exactly \( n \) clock periods are required to clock the packet back to register 1, and only \( R[t_s/\rho] \) clock periods elapse before the packet is clocked into a processor). Corollary to this is the fact that packets are supplied at every clock period.

* \( R[j/k] \) denotes the remainder of the division \( j/k \).
The second result is that only processors 1 through \( p \) will ever begin execution. Hence a system need have only \( p \) processors. If we add a fourth processor to the system of figure 3-1 it is clear that its input register will receive no packets. After processors 1, 2, and 3 begin execution initially, they become free to take new packets before packets can propagate to input register 4 (since \( p = 3 \)).

The case in which \( p > n \) is considerably more complicated since some or all of the input packets recirculate. This prevents packets from being supplied with every clock pulse. The following lemma quantifies the times at which packets can be supplied to the input loop:

**Lemma 3:** Assume \( p > n \). After the first \( n \) clock pulses then packets will be supplied at the following times only: \( n + \hat{i} - 1 + j p \) where \( \hat{i} = 1, \ldots, n \) and \( j \in \mathbb{Z}^+ \). Denote this (infinite) set of supply times as \( T_S \).

The proof of Lemma 3 is a straightforward application of the definition of the system. Packets are supplied at time \( t_s \) if either input register \( n \) contains a blank at time \( t_s - 1 \) or processor \( n \) signals that it will remove at time \( t_s \) the packet residing in input register \( n \). Blanks occur in input register \( n \) when one of the processors 1 through \( n-1 \) takes a packet and the blank thus produced propagates to latch \( n \) (since the input loop is initially empty, there are \( n \) supply times not covered by this analysis; hence the qualification in the statement of the lemma). Processor \( n \) signals the supply device at each of its initiation times. These two con-
ditions for the supply of a packet to the input loop can be used together with Lemma 2 to generate the desired formula. The formula indicates that at most \( n \) packets can be supplied every \( \rho \) clock periods; moreover, they are supplied at contiguous times. These facts can be used to relax assumptions 1 and 2 concerning the supply device. It need only generate \( n \) packets every \( \rho \) clock periods as long as they are ready when needed by the input loop. Figure 3-2 shows the behavior of a system with \( n = 3 \) and \( \rho = 4 \). After 3 clock pulses (at times 0,1,2), packets are supplied at times 3,4,5 and 7,8,9, etc. That is, three packets are supplied every 4 clock periods, as predicted. Further, the packets supplied at times 3,4,5 correspond to values of \( \hat{i} = 1,2,3 \) where \( j = 0 \); the packets supplied at times 7,8,9 correspond to values of \( \hat{i} = 1,2,3 \) where \( j = 1 \), and so on.

A parallel theorem to Theorem 1 for the case \( \rho > n \) can now be stated:

**Theorem 2:** Assume \( \rho > n \). Then if a packet is supplied at time \( t_s \in T_s \), then it will be taken from the input loop as follows:

If \( \rho > n + \hat{i} \), then processor \( n + \hat{i} - \rho \) will take it at time \( n + \hat{i} - \rho + t_s \); otherwise, processor \( n - R((\rho-n-\hat{i})/n) \) will take it at time \( n - \hat{i} + \rho - 2R((\rho-n-\hat{i})/n) + t_s \), where \( \hat{i} = R((t_s-n)/\rho) + 1 \).

The proof is similar to that of Theorem 1. Given the supply time, \( t_s \), of the input packet (picked from the set of legal supply times for the system), the location of the packet in the input loop at all subsequent times is known. Lemma 2 gives all processor initiation times, and again (as in the proof of Theorem 1) the
**SYSTEM BEHAVIOR: n = 3, \rho = 4**

<table>
<thead>
<tr>
<th>TIME</th>
<th>SYSTEM</th>
<th>TIME</th>
<th>SYSTEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>- - -</td>
<td>6</td>
<td>- 1 -</td>
</tr>
<tr>
<td></td>
<td>- - -</td>
<td></td>
<td>5 2 3</td>
</tr>
<tr>
<td></td>
<td>1 - -</td>
<td></td>
<td>4 6 -</td>
</tr>
<tr>
<td>1</td>
<td>- - -</td>
<td>7</td>
<td>- 2 1</td>
</tr>
<tr>
<td></td>
<td>1 - -</td>
<td></td>
<td>5 6 3</td>
</tr>
<tr>
<td></td>
<td>2 - -</td>
<td></td>
<td>7 4 -</td>
</tr>
<tr>
<td>2</td>
<td>- - -</td>
<td>8</td>
<td>- - 2</td>
</tr>
<tr>
<td></td>
<td>1 - -</td>
<td></td>
<td>5 6 3</td>
</tr>
<tr>
<td></td>
<td>3 2 -</td>
<td></td>
<td>7 4 -</td>
</tr>
<tr>
<td>3</td>
<td>- - -</td>
<td>9</td>
<td>5 - 3</td>
</tr>
<tr>
<td></td>
<td>1 2 -</td>
<td></td>
<td>8 6 4</td>
</tr>
<tr>
<td></td>
<td>4 3 -</td>
<td></td>
<td>9 - 7</td>
</tr>
<tr>
<td>4</td>
<td>- - -</td>
<td>10</td>
<td>- 5 -</td>
</tr>
<tr>
<td></td>
<td>1 2 -</td>
<td></td>
<td>8 6 4</td>
</tr>
<tr>
<td></td>
<td>5 4 3</td>
<td></td>
<td>7 9 -</td>
</tr>
<tr>
<td>5</td>
<td>1 - -</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5 2 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3-2**
first processor capable of taking the packet and the time at which
this occurs can be determined. Unlike Theorem 1, the solution
requires that the proof be broken into two cases. For $\rho < n + \hat{i}$,
the packet does not recirculate but for $\rho \geq n + \hat{i}$ it must before
a processor can take it. The definition of $T_s$ causes $\hat{i}$ to only
take on the values $1,\ldots,n$ and thus for $\rho \geq 2n$, all packets must
recirculate. Figure 3-2 shows that of the three packets supplied
at times 3,4,5 only the packet supplied at time 3 (number 4)
recirculates. Since the three packets correspond to values of
$\hat{i} = 1,2,3$ respectively, $\rho < n + \hat{i}$ for $\hat{i} = 2,3$ and hence these
packets do not recirculate. Packet 4 is taken by processor
$3 = n + \hat{i} - \rho + \tau_s = 3 + 1 - 4 + 3$ at time
$9 = n - \hat{i} + \rho - 2R[(\rho-n-\hat{i})/n] + \tau_s = 3 - 1 + 4 - 2R[(4-3-1)/3] + 3$;
packet 5 (supplied at time 4) is taken by processor
$1 = n + \hat{i} - \rho = 3 + 2 - 4$ at time $5 = n + \hat{i} - \rho + \tau_s = 3 + 2 - 4 + 4$;
packet 6 (supplied at time 5) behaves accordingly. Figure 3-3
shows the behavior of a system with $n = 3$ and $\rho = 6$. Every packet
supplied after time 2 recirculates before it is taken by a processor.

Theorems 1 and 2 should yield the same results when $\rho = n$.
Substituting $\rho$ for $n$ in Theorem 2 yields a value of
$\hat{i} = R[(\tau_s-\rho)/\rho] + 1 = R[\tau_s/\rho] + 1$. Since $\rho < \rho + R[\tau_s/\rho] + 1$ for
all times $\tau_s > 0$, then processor $\rho + \hat{i} - \rho = \hat{i} = R[\tau_s/\rho] + 1$ takes
every input packet at time $\rho + \hat{i} - \rho + \tau_s = \tau_s + R[t/\rho] + 1$ which
is in complete agreement with Theorem 1. Note that when $\rho = n$,
Lemma 3 indicates that packets are supplied at every clock pulse.
THEORETICAL SYSTEM BEHAVIOR: P, T VERSUS N WITH $\rho = 10, \beta = 0$

**FIGURE 3-4**

- P
- T

number of processors, n

P

18 17 16 15 14 13 12 11 10 9

0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
Clearly, packets will be supplied to the system at the first \( n \) clock times \( 0, \ldots, n-1 \). These \( n \) packets will be taken by the \( n \) idle processors. For all other supply times, Theorem 2 says that the supplied packets will be taken in finite time \((\rho \geq n)\). Theorem 1 says that all supplied packets will be taken in finite time as well \((\rho \leq n)\). Hence, for any value of \( \rho \), no packet can get trapped on the input loop and circulate forever (denote this situation as a "deadlock"). This freedom from deadlock is crucial to the value of the system.

System "throughput", \( T \), is defined here as the number of tasks processed by the system (that is, the number of tasks accepted by the acceptor device) per clock period. The system "overhead", \( P \), for the processing of a task is defined as the amount of elapsed time (in clock periods) from when the task's input packet is created and ready to be supplied to when its output packet is accepted, minus the task execution time, \( \rho \). In general, the average value of \( P \) for all tasks routed through the system is of interest in evaluating its performance. Theorems 1 and 2 enable throughput and overhead to be formulated:

**Corollary 1.1:** Assume \( \rho \leq n \). Then \( P = n + 2 \) and \( T = 1 \).

This corollary should be evident from inspection. Since there is no recirculation, all packets need propagate through only \( n \) registers from the supply to the acceptor regardless of which processor takes them. This requires \( n + 1 \) clock pulses. One additional pulse is required to clock the packet into the processor, making the total \( n + 2 \). (The packet is clocked onto the output)

* Supply / accept times are taken as 0 in corollaries 1.1, 1.2.
loop precisely at the end of its execution time, \( \rho \), if there are no conflicts—there are none—and, hence, no time is required.) Hence \( P = n + 2 \). Since packets are supplied at every clock pulse and the system has finite storage capacity, then a packet must be accepted at every clock pulse (neglecting transient behavior). Hence \( T = 1 \).

Figure 3-1 illustrates that all packets in the system of three processors (with \( \rho = 3 \)) require 8 clock pulses to pass through the system (including one pulse to get onto the input loop). Subtracting task execution time, \( P = 8 - 3 = 5 \) which is precisely \( n + 2 \).

After time 5, packets stream off the output loop at every clock pulse, and thus \( T = 1 \).

**Corollary 2.1:** Assume \( \rho > n \). Then \( < P >^* = \rho (1 + \frac{1}{n}) + 1 \) and \( T = n/\rho \).

In general, \( P \) will vary from packet to packet; the average value of \( P \) for the \( n \) packets supplied in the \( \rho \) clock period interval (corresponding to the \( n \) distinct values of \( i \) and a given value of \( j \)) can be calculated, however. If the system is allowed to run for long times and if transient startup behavior is neglected (corresponding to the first \( n \) packets supplied), then this is a reasonable value for system overhead. The proof separates \( < P > \) into average values for

a) The time required to supply a packet to the system
   \[ = \gamma_5, \]

b) the time required to clock the packet from input register 1 into a processor \[ = \gamma_1, \] and

\(^* \langle P \rangle \) denotes the average value of \( P \) taken for the \( n \) packets supplied in a time interval of \( \rho \) clock pulses.
c) the time required to clock a packet from the processor's output register into the acceptor = \( \gamma_0 \).

(All times stated in clock periods).

Now, \( < \gamma_S > = \rho/n \) since only \( \rho \) packets are supplied every \( n \) clock periods. \( < \gamma_i > = \rho + 1 - \frac{1}{2} (n+1) \), calculated from the formulas of Theorem 2. (That the two cases in Theorem 2 yield the same result for \( < \gamma_i > \) is remarkable.) The value of \( < \gamma_0 > = \frac{1}{2} (n + 1) \) since the \( n \) packets are processed in \( n \) separate processors. Combining the above, \( P = \rho(1 + 1/n) + 1 \), as stated. \( T = n/\rho \) by the same reasoning as in Corollary 1.1; since \( n \) packets can be supplied every \( \rho \) clock pulses, exactly \( n \) packets must be accepted in the same amount of time or else packets would accumulate in the system.

The three packets supplied at times 7, 8, and 9 in figure 3-2 require exactly 13, 9, and 9 clock periods in the system, respectively. This includes one period each for packets 8 and 9 (supplied at times 8 and 9) to be supplied; packet 7 (supplied at time 7) requires 2 clock periods to be supplied since it was created at time 5 (just after packet 6 was supplied) and could not be supplied until time 7. Subtracting off task execution time \( \rho = 4 \), \( P = 9, 5, \) and 5, respectively, and thus \( < P > = 19/3 = \rho(1 + \frac{1}{n}) + 1 = 4 (1 + \frac{1}{3}) + 1 \). The two corollaries should agree for \( n \). Substituting for \( \rho \) in Corollary 2.1, \( T = 1 \) and \( P = n(1 + \frac{1}{n}) + 1 = n + 2 \), which are of course the results of Corollary 1.1.

Figure 3-4 graphs the expressions for \( T, P \) versus the number of processors, \( n \). Several features should be noted. Throughput climbs linearly until \( \rho = n \) at which point it is constant for all
n \geq p. That is, a system may be expanded by adding processors with linearly improving throughput until \( p \) processors are in service, at which point it does not pay to add any more processors as they will not be used. The overhead decreases rapidly as processors are added to the system and reaches a minimum when \( p = n \). It then increases linearly for \( n > p \) since every output packet has to propagate through the registers of the unused, extra processors. It is fortunate that both \( P \) and \( T \) reach their optimal values for the same value of \( n (=p) \). With this configuration (denote it as the "optimal" system):

1) packets can be supplied at every clock pulse, and thereby throughput is maximal \((T = 1)\),

2) no packet recirculates in the input loop, and hence overhead is minimal and proportional to task execution time \((P = n + 2 = p + 2)\),

3) no packet can deadlock in the input loop and thus fail to get processed.

Considering the simplicity of the control structure (and keeping in mind the restrictions imposed), these results are encouraging.

The following observation can be made:

**Lemma 4**: If \( p \) is odd, then at most one processor will initiate execution at every clock pulse.

The proof of this lemma shows that, under the above restrictions, the set of processor initiation times (Lemma 2) maps into distinct integers in \( \mathbb{Z}^+ \). Figure 3-5 demonstrates this fact for \( p = 3 \). It also indicates that for \( p = 4 \) at most two processors initiate at every clock pulse (this is true for any even value of \( p \)).
FIGURE 3-5a: DISPLAY OF PROCESSOR INITIATION TIMES
\[ p = 3, n = 3; \ X = \text{processor initiation} \]

FIGURE 3-5b: DISPLAY OF PROCESSOR INITIATION TIMES
\[ p = 4, n = 3 \]
Note that if \( n = p \), a processor initiates with every clock pulse.

Assume that a single bus connects all processors with common memory (and/or devices). Now, break processor execution time into three segments as follows:

\[
\begin{align*}
&<------------- T_p ----------------> \\
&<----- T_b ----> <------------ T_p --------------><--- T_b ------>
\end{align*}
\]

where \( T_b \) corresponds to the time required for a bus access. \( T_b^1 \) allows the processor to fetch input data before execution; and \( T_b^2 \) allows it to store output data afterwards. The task execution time is now \( T_p \). Define \( \rho^* = \frac{2T_b + T_p}{T_c} = \rho + 2\beta \). (If \( \beta = 0 \), as before, the \( \rho^* = \rho \).)

**Theorem 3:** If \( \rho^* \) is odd and \( \beta = \frac{1}{2} \), then no processor will ever wait to use the bus.

The proof follows directly from Lemma 4. In the minimal one clock period between two processor initiations, the first half-clock period is reserved for the processor which just initiated a new task to access the bus and the second half-clock period is reserved for the processor which just completed its task and will initiate a new one when the clock period ends. Figure 3-6 illustrates this behavior. Processor 1 grabs the bus at time 4 when it initiates its second task; processor 3 grabs it just after processor 1 finishes with the bus at time 4.5. Note that processor execution time is two clock periods in order to make room for two bus accesses (together requiring one time unit) (\( \rho^* = 3 \)). In the optimal system the bus is continually in use.
FIGURE 3-6: DISPLAY OF PROCESSOR INITIATION TIMES WITH NO BUS CONFLICTS;
\[ \rho' = 3, \beta = 1/2, n = 3 \]
The above bus allocation scheme is corollary to the analysis of the control structure and is only meant to illustrate that data flow structures can be incorporated into the system without disrupting its behavior. The analysis serves properly to show that the loop structured technique is a viable scheme for the effective control of a multiprocessor, especially in situations in which its simplicity is advantageous.
Chapter 4
THE SIMULATION

In order to verify the analysis of the loop structured system and to determine its behavior as the restrictions imposed by the analysis are lifted, a simulation program was written. The simulator, coded in PL/I, is composed of seven software constructs interacting as concurrent processes synchronized by a system clock. The synchronization and simulation of a system clock is handled by a seventh construct called the "process sequencer". The process sequencer activates each of the processes at simulated time 0; each process eventually suspends its execution by passing parameters to the sequencer signifying the number of clock pulses that must pass before execution can resume ("wait-time") and at what location (PL/I label) this execution should take place. The sequencer determines the next earliest time at which one or more processes can resume execution, advances the system clock to this time, and sequentially activates the processes. (The order of activation is significant, as will be explained later.) When all such processes have again suspended execution, this algorithm is repeated unless one of two criteria have been satisfied, as established by parameters:

1) the system clock has reached a maximum allowable value, or

2) a desired number of tasks have been accepted.

The processes can communicate with one another in a scheme which adopts the concepts of Dijkstra's semaphores and primitive operations P and V for the synchronization of asynchronous parallel
processes [3]. A process suspends execution indefinitely, contingent on the action of another process, by decrementing a semaphore and suspending execution with the longest allowable wait-time passed to sequencer. As a second appropriate process determines that the first one should be reactivated (by incrementing the semaphore and observing its value), it instructs the sequencer, with a special command, to reactivate the process at the earliest possible time.

The seven concurrent processes will be discussed separately; flow charts can be found in figure 4-1.

1) the supply process: This process simulates the actions of the supply device by sequentially generating tasks, constructing input packets, and passing the packets to the input loop (discussed later). The input packets are constructed with the following fields:

   a) a one bit tag indicating the presence of data,
   b) a task identification number,
   c) the task execution time,
   d) the number of data values to be input from external memory, and
   e) the number of data values to be output to memory.

The values for these fields are determined by parameters passed to the system when the simulation commences. In particular, fields c, d, and e can be generated as fixed values for the entire simulation or random values between two limits. (Of course, a more complicated scheme, such as having exactly half as many output values as input values, could be implemented with simple changes
SIMULATOR FLOWCHARTS

SUPPLY PROCESS

ACCEPTOR PROCESS

FIGURE 4-1a
at compile time.) Parameters instructing exactly how much time is to be spent in each of the supply processes's duties must also be supplied (see table 4-1).

2) the processor process: This process is coded as a re-entrant software construct, each entry representing a different processor in the system. The process (1) interrogates its input register for an input packet, (2) inputs data values by communication with the bus process (discussed later), (3) "performs" the task by suspending execution for the task execution time, (4) outputs data values by another interchange with the bus process, (5) generates an output packet, (6) and deposits same in its output register when possible. The output packet has the following fields:

   a) a one bit tag indicating the presence of data, and
   b) the task identification number.

Requests for the use of the bus are made by incrementing a processor/bus semaphore and activating the bus process if necessary. The processor calculates when it would be able to use the bus by observing the value of the semaphore and knowing bus access time, a system parameter. It then suspends execution until the time at which it would complete its bus access in a real system. (Of course, in this simulated environment, no actual passing of data or execution of tasks takes place.) There is a communication link between the supply process and processor process(es). Whenever the supply puts a packet on the input loop, it increments a semaphore; the processor that removes the packet decrements the
SIMULATOR PARAMETERS

1. number of processors
2. minimum task execution time
3. allowable range for random task execution times
4. minimum number of input (output) operands
5. allowable range for random number of operands
6. number of operands obtainable with one bus access
7. bus access time
8. supply device packet construction time
9. acceptor device packet decoding time
10. system clock time
11. maximum simulation time
12. maximum number of processed tasks in one simulation

TABLE 4-1
semaphore. Whenever the input loop is empty, the idle processors (those waiting for new tasks) suspend execution indefinitely; the supply activates them when necessary. This interchange greatly enhances the efficiency of the simulation.

3) The acceptor process: Output packets are removed from the output loop by this process, which simulates the actions of the acceptor device. The process suspends execution for a fixed time (specified by a system parameter) to simulate packet decoding and system table updating functions. A communication link between the processor process(es) and the acceptor process exists to allow the acceptor to suspend execution when it is waiting for an output packet and the output loop is empty; the appropriate processor activates the acceptor when the next output packet is deposited on the loop.

4) The input loop process: This process simulates the action of the input loop by updating a pointer into the PL/I array containing the input packets. Processors and the supply select the proper array location in which to fetch and store packets by using offsets to this pointer. As an efficiency measure, instead of updating this pointer at each clock pulse, the process determines the next time of any process activation from the sequencer and adjusts the pointer to make the input loop appear as it should at that time; it then suspends execution until then. (This, of course, requires that the input loop process be activated by the sequencer just after the supply, processors, and acceptor have suspended execution at every given time.)
5) **The output loop process:** This process precisely parallels the operations of the input loop and no more will be said of it here.

6) **The bus process:** The simulator incorporates the use of a single bus connecting all processors for the passing of data to and from memory. The effects produced by this process can be negated by setting the parameter for bus access time to zero. This process decrements the bus communication semaphore (previously described) and waits for one bus access time or suspends execution indefinitely, depending on the semaphore's value (which indicates whether a processor is waiting for the bus).

7) **The monitor process:** This process produces snapshots of the system at regular intervals as determined by a system parameter. Although not a part of simulated system, it is treated like any other process.

The special signaling between processor n and the supply, and between the acceptor and processor 1 (described in Chapter 2), is implemented by having the sequencer properly order the activation of the processes, as follows: (1) the acceptor, (2) all processors, and (3) the supply. The acceptor can thus remove a packet from output register n before processor 1 can inspect it, and processor 1 can remove a packet from input register n before the supply can inspect that register.

A software construct gathers statistics during the course of the simulation and presents them upon termination. The values for throughput and overhead (broken down into its constituent parts as
described in Chapter 3) are given as well as the following:

1) processor utilization expressed as the percentage of time engaged in waiting for input packets, waiting for the bus, executing tasks, and waiting for a space to deposit output packets.

2) loop utilization expressed as the percentage of loop registers actually containing data during the course of the simulation (given for each loop).

The simulator correctly verified the results of the preceding chapter within certain tolerances. More precisely, the graphs of figure 4-2 produced by the simulator have minor perturbations due to (a) the transient behavior at startup (when packets are supplied at every clock pulse) which influenced system overhead and (b) the finite execution time of the simulator which influenced throughput.

Figure 4-2 graphs $P$ and $T$ versus the number of processors, $n$, where $\rho = 8$. The simulator ran for 500 clock pulses for each of the values of $n$ (1, 2, 4, 6, 8, 10, 12, 14, 16). Theorem 3 (absence of bus conflicts when $\rho$ is odd and $\beta = \frac{1}{2}$) was verified as well; see figure 4-3, where $P$ and $T$ are plotted versus $n$ for $\beta = 0, \frac{1}{2},$ and 1 and $\rho = 10$. The simulator verified that when $\beta = \frac{1}{2}$, no bus conflicts occurred ($\rho^* = 11$). The curves for $P$ and $T$ reflect this fact. In general, the bus will have an effect on system throughput and overhead as the curves with $\beta = 1$ indicate. The overhead and throughput have degraded considerably, although the shape of the curves corresponds amazingly well with the others. Both curves reach optimal values for the same number of processors ($n = 6$). With more processors on line,
THROUGHPUT AND OVERHEAD VERSUS N WITH $\rho = 8$

FIGURE 4-2
EFFECT OF BUS CONFLICTS: $T$ VERSUS $N$ WITH $\rho=10$, $\beta=0$, $1/2$, $1$

**Figure 4-3a**
EFFECT OF BUS CONFLICTS: P VERSUS N WITH $\rho=10$, $\beta=0$, $1/2$, 1

\[ \text{FIGURE 4-3 b} \]
the number of bus conflicts prevents throughput from increasing, and overhead climbs quickly (although linearly).

The restriction that less than one clock period be required to construct a packet in the supply device and decode it in the acceptor device can be lifted and its effect on throughput and overhead observed. Figure 4-4 shows $P$ and $T$ versus $n$ for the above parameters equal to 0, 1, 2, and 3 clock periods and $\rho = 10$. The increase in supply/acceptor times seems to move the curves to the left and down on the graphs, indicating that fewer processors were required to handle the reduced load produced (as expected). Throughput suffers, of course, but overhead is reduced if only the required processors are used. Again, the curves retain the shape predicted by the analysis when a restriction is lifted and both $P$ and $T$ reach optimal values for the same number of processors on corresponding curves.

Figure 4-5 reveals system behavior when the restriction of fixed task execution times ($\rho$) was lifted. Tasks were supplied with an even distribution around the mean value of 10; only integer task times were allowed. The three sets of curves were produced when $\rho$ was allowed to vary as much as $\pm 1$, $\pm 2$, and $\pm 3$ clock periods on separate tasks. All curves for $P$ are amazingly similar, and those for $T$ are identical. Although $\rho$ varied only 30% from its mean value, the effect of random task times was minimal. The system behaved as if task execution times were fixed at this mean value.

Finally, a system was simulated in which all of the above restrictions were simultaneously lifted (figure 4-6). With $\rho = 10 \pm 2$, $\beta = 1/2$, and supply/acceptor times = 2 (clock periods), the overhead
EFFECT OF NON-ZERO SUPPLY / ACCEPT TIMES: T VERSUS N WITH ρ=10, β=0

Figure 4-4a

- S/A TIME = 0.1 (clock periods)
- S/A TIME = 2
- S/A TIME = 3
EFFECT OF FINITE SUPPLY / ACCEPT TIMES: P VERSUS N WITH $p=10$, $\beta=0$

FIGURE 4-4 b

S/A TIME = 3 → + S/A TIME = 2
+ S/A TIME = 0.1
(clock periods)

N

P

2 4 6 8 10 12 14 16
EFFECT OF RANDOM TASK TIMES: $P, T$ VERSUS $N$ WITH $\rho = 10^1, 2, 3$ AND $\beta = 0$

FIGURE 4-5
SAMPLE FULLY GENERAL SYSTEM: T, P VERSUS N WITH $\rho=10\pm2$, $\beta=1/2$, S/A TIME=2
oscillated before climbing quickly after \( n = 10 \). The throughput climbed linearly and leveled off at \( T = 0.5 \) after \( n = 7 \). The data points are included in the graph since the curves drawn are subject to interpretation. The curves vaguely resemble theoretical behavior, but clearly, complex interactions are taking place in the system. A more sophisticated analysis is required for an understanding of the loop structured system in its full generality.

The simulator is useful in determining the behavior of the system as restrictions are lifted. The above investigations are quite limited, and future efforts could include:

1) random task arrival times at the supply device (namely, arrivals with a Poisson distribution),
2) random processor failure/repair with some mean time to failure/repair,
3) the introduction of extra latches in the loops (possibly serving to better buffer the randomly varying system loads).
4) the introduction of more complex data flow paths (e.g., multiple buses or even the passing of data on the loops) with random demands made on such paths.

Clearly, any study of the control structure in a realistic environment requires the introduction of a stochastic analysis. The simulator would provide a useful vehicle for such analysis.
EXTENSIONS AND CONCLUSIONS

A loop structured system for the distribution and collection of control information in a multiprocessor has been described, analyzed, and simulated. The scheme makes effective use of the processing resources of the system and behaves in a desirable manner under the various conditions and restrictions specified in the preceding chapters. Many problems peripheral to this discussion but critical to the realization of such a system must be investigated and solved. These include the design of the supply and acceptor devices, the formulation of tasks and characterization as packets, determination of optimal memory allocation strategies and data flow structures, etc.

Alternatives to the basic design should also be considered. The use of a single input loop with one entry point favors the processors closest to that entry point (input register 1); all input packets must pass through input register 1 regardless of processor 1's state (busy or idle). Perhaps two loops clocking packets in opposite directions would more efficiently deliver tasks to waiting processors. The supply could alternatively deposit packets on different loops, and a processor could select a packet from either of its two input registers. (See figure 5-1.) There is added complexity with this technique; the characteristics of the system would dictate its relative merits. Possibly a processor could dynamically remove its register from the loop while it is performing a task. The supply would then deposit its packet in the
A TWO-LOOP MULTIPROCESSING SYSTEM

(output loop not shown)

S = supply device
I.R. = input register
P = processor

FIGURE 5-1
register of the first processor capable of taking it (at the following clock pulse). If all processors were busy, then the supply would be inhibited from depositing the packet at all. Thus, packets would be directed more quickly to an available processor, naturally, at the cost of a more complex control structure.

A radical departure from the basic design would eliminate the output loop altogether. Since the output packets direct the acceptor to update system tables which are used by the supply to construct more tasks, then perhaps the processors could construct new input packets themselves. A single loop could be used; a processor could remove packets from its loop register when necessary and deposit new packets in the register it its currently running task spawned others. If required, a special purpose device could be attached to the loop with the job of updating system tables. It would look for specially tagged packets directed to it from the processor, instructing it to make a particular update. Figure 5-2 depicts this system. Relatively powerful processors are envisioned for this system as their duties would be complex. This scheme looks very similar to the data communication loops of Farber and others (see Fraser for a discussion of these techniques)[5]. There is an important difference; packets are not directed to particular processors (as "messages") but are available to the first processor to take them (with the possible exception of the special device). Processors are anonymous in this system.

The foregoing results indicate that the loop structured technique is a useful alternative to other methods of multiprocessor
A SINGLE LOOP MULTIPROCESSOR

R = register
P = processor

FIGURE 5-2
control. Further research into this and related strategies has been shown to be warranted.
Lemma 1: Processor $i$ begins execution initially at time $2i-1$, for $i = 1, \ldots, \min(p, n)$.

Proof: Prove by induction on $n$, the number of processors in the system. Basis step: $n=1$. The lemma is trivially true; all processors are initially idle, and hence the first packet supplied to the system (at time 0) is taken by processor 1 at time 1.

Induction step: Assume the statement of the lemma for $n$ processors and show that it holds for $n+1$ processors. Note that the addition of another processor to the system (processor $n+1$) does not affect the initial behavior of processors 1, $\ldots$, $n$.

CASE I: $n<p$. Here $\min(p, n) = n+1$, and we must show that processor $n+1$ initiates at time $2(n+1)-1$. A packet, $x$, was delivered to register 1 at time $n$ (since packets are supplied at all clock pulses). This packet was not taken by any processor $k$ where $k=1, \ldots, n$ because it arrived at register $k$ at time $n+k-1$ and could only have been taken by processor $k$ at time $n+k$. Now, processor $k$ began execution at time $2k-1$ (by the induction hypothesis) which is clearly less than $n+k$ for any $k$. Processor $k$ will remain busy for at least $p$ clock periods and cannot possibly take the packet until time $2k-1+p$ which is greater than $n+k$ since $n<p$. Hence no processor $k$ can take packet $x$ and it arrives in register $n+1$ at time $2n$. Processor $n+1$ will take the packet and begin execution at time $2n+1 = 2(n+1) - 1$ if it is not busy at this time. Since processor $n$ begins execution initially at time
2n-1 (induction hypothesis), then no packet could have reached register n before time 2n-2 or else processor n would have initiated earlier. Processor n+1 cannot begin execution for two more time units after processor n does, which is time 2n+1 = 2(n+1) - 1. Thus, packet x is taken by processor n+1 when it begins execution at time 2(n+1)-1.

CASE II: \( n > p \). Here \( \min(p, n+1) = p \), which is the same as \( \min(p, n) \).

Hence, processor n+1 is not claimed to initiate by the statement of the lemma and there is nothing to be proved in this case. Thus, processor i begins execution initially at time \( 2i-1 \) for \( i=1, \ldots, \min(p, n) \).

**Lemma 2:** Processor i will begin execution only at times \( 2i-1+jp \), where \( i=1, \ldots, \min(p, n) \) and \( j \in \mathbb{Z}^+ \). That is, processor i, once started, is always busy.

**Proof:** Prove by induction on j, the number of activations of each processor.

**Basis step:** \( j=0 \). The lemma holds trivially by lemma 1.

**Induction step:** Assume the lemma is true for all j such that \( j \leq J \).

That is, for any \( n, p \in \mathbb{Z}^+ \), processor i will initiate execution at times \( 2i-1+jp \) only for \( i=1, \ldots, \min(p, n) \) and \( j = 1, \ldots, J \). Show that processor i will initiate at time \( 2i-1+(J+1)p \) by induction on i.

**Basis step:** \( i = 1 \). Processor 1 started execution at time \( 1 + Jp \) (outer induction hypothesis) and hence, completed execution at time \( 1 + (J + 1)p \) since all tasks take \( p \) clock periods. Because no packets recirculate, it was not delayed in depositing its output packet and thus was ready to clock in a new input packet at this
time. An input packet is in register 1 at all clock periods and, in particular, at time $1 + (j+1)\rho$. Hence processor 1 began execution at this time.

Induction step: Assume processor $i$ begins execution at time $2i - 1 + (j+1)\rho$ for all $i \leq \bar{T} < \min(\rho, n)$. Show that processor $\bar{T} + 1$ begins execution at time

$$2(\bar{T}+1) - 1 + (j+1)\rho.$$ 

A packet, $x$, was delivered to register 1 at time $\bar{T} + (j+1)\rho$ by the packet supply. This packet was not taken by processor $i$ because it arrived in register $i$ at time $t_1 = \bar{T} + i + (j+1)\rho$.

Processor $i$ began execution at time

$$2i - 1 + (j+1)\rho$$

which is clearly less than $t_1$ for any $i = 1, \ldots, \bar{T}$. Processor $i$ will remain busy for at least $\rho$ clock periods and cannot possibly take the packet until

$$2i - 1 + (j+2)\rho$$

which is clearly greater than $t_1$ for $i = 1, \ldots, \bar{T}$ since $\bar{T} < \min(\rho, n) \leq \rho$. Hence no processor $i$ can take packet $x$ and it arrives in register $\bar{T} + 1$ at time

$$2\bar{T} + (j+1)\rho.$$ 

Processor $\bar{T} + 1$ will take the packet at time

$$2\bar{T} + 1 + (j+1)\rho = 2(\bar{T}+1) - 1 + (j+1)\rho.$$
if it is ready to begin execution at this time. Now, processor T + 1 began execution at time 2(T+1) - 1 + Jp (outer induction hypothesis) and is ready to deposit its output packet at time 2T + 1 + (J+1)p since all tasks take p time units. It is not delayed because any delay would be caused by a packet in its output register at this time. Such a packet would have been deposited on the output loop by some processor i exactly T + 1 - i clock periods previously (since no packets recirculate) or at time

\[ t_2 = 2T + 1 + (J+1)p - (T+1-i) = T + i + (J+1)p. \]

Since processor i last initiated at time

\[ 2i - 1 + (J+1)p \]

(inner induction hypothesis), which is less than \( t_2 \) (\( i \leq T \)), and cannot initiate again for \( p \) clock periods, at a time greater than \( t_2 \) (\( T < p \)), then no processor i could have deposited a packet on the output loop such as to delay processor T + 1 at time 2T + 1 + (J+1)p. Hence processor i + 1 takes packet x at this time, which was to be proved.

**Theorem 1:** Assume \( p \leq n \). Then if an input packet is supplied at time \( t_S \), processor R\((t_S,p) + 1 \) will take it from the input loop at time \( t_S + R\((t_S/p) + 1 \).

**Proof:** Pick arbitrary time \( t_S \in \mathbb{Z}^+ \). Now \( t_S = Q(t_S/p) \cdot p + R\((t_S/p). \)

Let \( j = Q(t_S/p) \) and \( k = R\((t_S/p). \) Then \( j \in \mathbb{Z}^+ \) and \( k \in \{0,...,p-1\}. \)

Assume a packet, x, is supplied at time \( t_S \). Suppose the packet is taken by some processor i. Then it arrives at processor i's input register at time
where $i \in \{1,...,n\}$ and $\ell \in Z^+$. Note that $\ell$ represents the number of times the packet must recirculate before it is taken by a processor. The packet is thus taken at time

$$t_s + \ell n + i$$

which is one of processor $i$'s initiation times

$$2i - 1 + j'p , \quad j' \in Z^+$$

according to lemma 2. Since the packet is taken by the first processor capable of taking it, then we must find the smallest integer $\ell n + i$ and any $j'$ such that

$$t_s + \ell n + i = 2i - 1 + j'p.$$  

Now, $t_s = jp + k$ so that

$$jp + k + \ell n + i = 2i - 1 + j'p.$$  

Pick $j' = j$, $i = k + 1$, and $\ell = 0$. Clearly, the equation is satisfied for these values of $j'$, $i$, and $\ell$, and $i \in \{1,...,n\}$. Since $\ell = 0$ then we must show that $i$ is the smallest integer satisfying the equation. Assume there exists $i'$ less than $i$. Then,

$$jp + k + i' = 2i' - 1 + j'p$$

and thus

$$i' = (j-j')p + k + 1 < i = k + 1$$

This implies that $(j-j')p$ is less than zero. Since $j$, $j'$, and $p$ are elements of $Z^+$, then

$$(j-j')p \in \{-p, -2p, ...\}.$$  

Now $k + 1 \leq p$ and hence $i' \leq 0$; thus $i'$ is not an element of $\{1,...,n\}$, and $i$ must be minimal. Finally, processor $i = k + 1 = R[t_s/\phi] + 1$ takes packet $x$ at time
\[ t_s + \ln + i = t_s + R[t_s/p] + 1, \]

as was to be proved.

**Lemma 3:** Assume \( p \geq n \). After the first \( n \) clock pulses, then a packet will be supplied at the following times only: \( n + i - 1 + j \rho \) where \( i = 1, \ldots, n \) and \( j \in \mathbb{Z}^+ \). Denote this (infinite) set of supply times as \( T_s \).

**Proof:** Packets can be supplied to register 1 at a time \( t \) if register \( n \) contains a blank at time \( t-1 \). Now, a blank occurs in register \( n \) whenever some processor \( i \) initiates execution by taking an input packet, which produces a blank in register \( i+1 \) that must propagate to register \( n \) (\( i = 1, \ldots, n-1 \)). Since the processor \( i \) initiates execution at times \( 2i-1+j \rho \) (lemma 2), then there will be a blank in register \( n \) at the following times:

\[ T_1 = 2i - 1 + j \rho + n - i - 1 \quad j \in \mathbb{Z}^+ \]

where \( n-i-1 \) represents the time required for the blanks to propagate to register \( n \). Thus packets can be supplied at times:

\[ T_{1s} = 2i - 1 + j \rho + n - i. \]

Packets can also be supplied at a time \( t \) when there exists a packet in register \( n \) at time \( t-1 \), and processor \( n \) signals the supply that it will take the packet at time \( t \). This occurs exactly at the times that it initiates execution:

\[ T_{2s} = 2n - 1 + j \rho \quad j \in \mathbb{Z}^+. \]

Together

\[ T_s = T_{1s} \cup T_{2s} = n + i - 1 + j \rho \]
where \( i = 1, \ldots, n \) and \( j \in \mathbb{Z}^+ \).

Since \( \rho > n \), then distinct integers are produced for each value of \( i \) and \( \rho \). Since the input loop is initially empty, packets are supplied on all of the first \( n \) clock pulses. After this time, packets can only be supplied at times in \( T_s \), according to the definition of the system. Hence the lemma is proved.

**Theorem 2:** Assume \( \rho > n \). Then if a packet is supplied at time 
\( t_s \in T_s \), then it will be taken from the input loop as follows:
If \( \rho > n + \hat{i} \), then processor \( n + \hat{i} - \rho \) will take it at time 
\( n + \hat{i} - \rho + t_s \); otherwise, processor \( n - R[(\rho-n-\hat{i})/n] \) will take it 
at time \( n - \hat{i} + \rho - 2R[(\rho-n-\hat{i})/n] + t_s \), where \( \hat{i} = R[(t_s-n)/\rho] + 1 \).

**Proof:** Pick arbitrary supply time \( t \in T_s \). Now
\[ t = n + \hat{i} - 1 + j \rho \]
for some \( i \in \{1, \ldots, n\} \) and \( j \in \mathbb{Z}^+ \).

The packet, \( x \), supplied at time \( t \), will circulate in the loop until 
a processor removes it from the loop by initiating execution.
Assume processor \( i \) takes packet \( x \). Then packet \( x \) will arrive at 
latch \( i \) at time
\[ t + \lambda n + i - 1, \]
where \( \lambda \in \mathbb{Z}^+ \) and \( i \in \{1, \ldots, n\} \)
and be taken by processor \( i \) at time
\[ t + \lambda n + i, \]
where \( \lambda \) represents the number of times packet \( x \) recirculates in the 
input loop.
Processor i initiates execution at times
\[ j'_p + 2i - 1 \quad j' \in Z^+ \]
by lemma 2
and hence we must find the smallest integer \( \lambda n + i \) and any \( j' \) such that
\[ j'_p + 2i - 1 = t + \lambda n + i. \]
Substituting for \( t \) we have:
\[ j'_p + 2i - 1 = n + j - 1 + j_p + \lambda n + i, \]
and thus:
\[ \lambda = [(j' - j) + i - (n + i)] / n \]

CASE I: \( \rho < n + \hat{i} \). Then set: \( j' = 1 + j \) and \( i = n + \hat{i} - \rho \) so that \( \lambda = 0 \).
Since \( n + \hat{i} > \rho \geq n \), we have \( i > 0 \) and \( i \leq \hat{i} \). Hence, \( i \in \{1, \ldots, \hat{i}\} \).
Since \( \hat{i} \leq n \) then \( i \in \{1, \ldots, n\} \) as it must be. Since \( \lambda = 0 \) and \( i \leq n \) we have a minimal value for \( \lambda n + i \) (see the proof of theorem 1).

CASE II: \( \rho \geq n + \hat{i} \).
Then set: \( j' = 1 + j \)
and \( i = n - R[n + \hat{i}] \)
which gives
\[ \lambda = 1 + Q[n + \hat{i}] \]

Clearly \( i \in \{1, \ldots, n\} \) since \( R[k] \in \{0, \ldots, n-1\} \) for any integer \( k \in Z^+ \).
We must show that \( \lambda n + i \) is minimal. Since \( i \in \{1, \ldots, n\} \) then we need only show that \( \lambda \) is minimal. Suppose there exists \( \lambda' < \lambda \).
Then
\[ \lambda' < \lambda = 1 + Q[n + \hat{i}] \]
\[ \lambda' < Q[n + \hat{i}] \]
Now
\[ \xi' = \frac{(i' - i)\rho}{n} + \frac{i' - (n+i)}{n} \]

where
\[ i' \in \{1, \ldots, n\} \]

Hence,
\[
0 \leq \xi' = \frac{i'}{n} + \frac{\rho-(n+i)}{n} + \frac{(j'-j-1)\rho}{n}
\]
\[
= \frac{i'}{n} + \frac{(j'-j-1)\rho}{n} + \frac{\rho-(n+i)}{n} + \frac{1}{n} R[p-(n+i)] \leq Q[\frac{\rho-(n+i)}{n}]
\]

which implies that
\[
\frac{i'}{n} + \frac{(j'-j-1)\rho}{n} + \frac{1}{n} R[p-(n+i)] \leq 0
\]

\[ \Rightarrow \]
\[
\frac{i'}{n} + \frac{1}{n} R[p-(n+i)] \leq -\frac{(j'-j-1)\rho}{n}
\]

Now \( 0 < \frac{i'}{n} \) since \( i' \in \{1, \ldots, n\} \)

and \( 0 \leq \frac{1}{n} R[p-(n+i)] \) since \( \rho > n + i \).

Thus \( 0 < \frac{i'}{n} + \frac{1}{n} R[p-(n+i)] \leq \frac{(j+1-j')\rho}{n} \)

Since \( \rho, n > 0 \) and \( j \in \mathbb{Z}^+ \) we must have
\[ j' < j + 1 \] and \( \frac{(j+1-j')\rho}{n} \geq \frac{\rho}{n} \)

Recall:
\[
0 \leq \frac{i'}{n} + \frac{\rho-(n+i)}{n} + \frac{(j'-j-1)\rho}{n}
\]

\[ \Rightarrow \]
\[
-\frac{(j'-j-1)\rho}{n} = \frac{(j+1-j')\rho}{n} \leq \frac{i'}{n} + \frac{\rho-(n+i)}{n}
\]

\[ \Rightarrow \]
\[
\frac{\rho}{n} \leq \frac{(j+1-j')\rho}{n} \leq \frac{i'}{n} + \frac{\rho-(n+i)}{n}
\]

\[ \Rightarrow \]
\[
\frac{\rho}{n} \leq \frac{i'}{n} + \frac{n}{n} - \frac{n}{n} \leq \frac{i}{n} \]

\[ \Rightarrow \]
\[ n \leq i - \hat{i} \]
Now, there does not exist \( i' \in \{1, \ldots, n \} \) such that for 
\[
i' \in \{1, \ldots, n \} \quad n < i' - \hat{i}.
\]
Hence there does not exist \( \varepsilon' < \varepsilon \), and hence \( \varepsilon \) is minimal.

In either case I or case II, packet \( x \) will be taken at time
\[
2i - 1 + j' \rho = 2i - 1 + j' \rho + t - (n + \hat{i} - 1 + j\rho)
\]
where
\[
t = \text{supply time (derived above)}
\]
\[
= n + \hat{i} - 1 + j\rho.
\]

Note that
\[
\hat{i} = R \left[ \frac{t - n}{\rho} \right] + 1
\]
since
\[
1 \leq \hat{i} \leq n \leq \rho.
\]
Hence we have seen that

if \( \rho < n + \hat{i} \) \quad \text{(Case I)}

then packet \( x \) will be taken at time
\[
2(n + \hat{i} - \rho) + (j' - j)\rho - n - \hat{i} + t = n + \hat{i} - \rho + t
\]
by processor \( \hat{i} = n + \hat{i} - \rho \);

otherwise packet \( x \) will be taken at time
\[
2(n - R[\frac{\rho - (n + \hat{i})}{n}]) + (j' - j)\rho - n - \hat{i} + t = n - \hat{i} + \rho - 2R[\frac{\rho - (n + \hat{i})}{n}] + t
\]
by processor \( \hat{i} = n - R[\frac{\rho - (n + \hat{i})}{n}] \).

The following two corollaries assume that packets can be constructed in zero time by the supply device. (This further restricts assumption 2 on page 12.)

**Corollary 1.1:** Assume \( \rho \leq n \). Then \( T = 1 \), and \( P = n + 2 \).

**Proof:** Since packets are constructed in zero time and supplied at every clock pulse (theorem 1), then all packets must wait one clock
period in the supply device before being clocked into register 1. Every task's input packet is taken by some processor i before recir-
culating (theorem 1), and hence, i clock pulses are required to clock the packet into the processor. Upon completion of the task, the processor deposits the task's output packet in output register i without delay (theorem 1); thus, exactly n-i+1 clock periods are required for the packet to reach the accepter. The total overhead, \( P \), for every task is then

\[
1 + i + n - i + 1 = n + 2
\]
as was to be proved.

Since packets are supplied at every clock period and storage capacity within the system is finite, then tasks must be completed at every clock period and hence, \( T = 1 \).

**Sub-lemma 1:** \[
\frac{1}{n} \sum_{i=1}^{n} R\left(\frac{m-i}{n}\right) = \frac{n-1}{2}
\]
where \( m \in \mathbb{Z}^+ \) and \( m > n \).

**Proof:**

\[
\Sigma\left(\frac{m-i}{n}\right) = \Sigma(Q \left[\frac{m-i}{n}\right] + \frac{1}{n} R\left[\frac{m-i}{n}\right])
\]

\[
= \Sigma Q \left[\frac{m-i}{n}\right] + \frac{1}{n} \Sigma R \left[\frac{m-i}{n}\right]
\]

\[
\frac{1}{n} \Sigma R \left[\frac{m-i}{n}\right] = \Sigma \left[\frac{m-i}{n}\right] - \Sigma Q \left[\frac{m-i}{n}\right]
\]

\[
= m - \Sigma \frac{i}{n} - \Sigma Q \left[\frac{m-i}{n}\right]
\]

Now \( \Sigma \frac{i}{n} = \frac{n + 1}{2} \).

Let \( m = kn + i' \) where \( 0 \leq i' < n-1 \).
Then \( Q \left[ \frac{m-i}{n} \right] = k \) for \( i'-i > 0 \),
that is, for \( i = 1, \ldots, i' \). Also,
\[
Q \left[ \frac{m-i}{n} \right] = k-1 \text{ for } i'-i < 0 ,
\]
that is, for \( i = i' + 1, \ldots, n \).
(Note, \( i'-i \geq -n \) since \( i < n \).)

Together,
\[
\sum_{i=1}^{n} Q \left[ \frac{m-i}{n} \right] = \sum_{i=1}^{i'} Q \left[ \frac{m-i}{n} \right] + \sum_{i=i'+1}^{n} Q \left[ \frac{m-i}{n} \right]
\]
\[
= i' \cdot k + (n-i')(k-1)
\]
\[
= kn + i'
\]
\[
= m - n
\]

Thus,
\[
\frac{1}{n} \sum R \left[ \frac{m-i}{n} \right] = m - \frac{n+1}{2} - m + n
\]
\[
= \frac{n + 1}{2}
\]
as was to be proved.

**Corollary 2.1:** Assume \( \rho > n \).

Then \( T = \frac{n}{\rho} \),
\[
\langle \rho \rangle = \rho \left( 1 + \frac{1}{n} \right) + 1
\]

**Proof:** Exactly \( n \) packets can be supplied to the system every \( \rho \) time units (Lemma 3). Since system storage capacity is finite, it must be the case that \( n \) tasks are processed every \( \rho \) time units, and hence \( T = n/\rho \).
Packets are supplied at the following times (from Lemma 3):

\[ T_S = n + \hat{i} - 1 + j \rho, \]

where \( j \in \mathbb{Z}^+ \) and \( \hat{i} = 1, ..., n \).

For each value of \( j \), \( n \) packets are supplied at \( n \) distinct clock pulses. An average overhead, \( < P > \), for the \( n \) tasks corresponding to these packets can be calculated:

\[ < P > = < \gamma_S + \gamma_i + \gamma_o >, \]

where

\( \gamma_S = \) elapsed time from packet construction to arrival on input loop,

\( \gamma_i = \) elapsed time from the input packet's arrival on input loop until its arrival in a processor, and

\( \gamma_o = \) elapsed time from the output packet's creation (at the time of task completion) until its arrival in the acceptor.

Further:

\[ < P > = < \gamma_S > + < \gamma_i > + < \gamma_o > \]

so we may calculate each average separately.

\( < \gamma_S > : \) The \( n \) packets are supplied on \( n \) contiguous time units (Lemma 3) whereupon there is a lapse of \( \rho - n + 1 \) time units until the cycle begins again. Since packets are created in zero time
(assumption), \( n - 1 \) of the packets have \( \gamma_S = 1 \) and 1 packet has \( \gamma_S = \rho - n + 1 \). Thus:

\[
< \gamma_S > = \frac{n-1}{n} \cdot 1 + \frac{1}{n} \cdot (\rho-n+1) = \frac{\rho}{n} .
\]

\(< \gamma_i > : If t_s is the time that packet \( \rho \) is supplied and \( t_e \) is the time that packet \( \rho \) is taken then:

\[
< \gamma_i > = < t_e - t_s > \text{ for all of the } n \text{ supplied packets corresponding to a value for } j. \text{ Each such packet has a corresponding unique value of } \hat{i} \text{ (Lemma 3). Hence there are two cases:}
\]

CASE I: \( \rho > 2n \)

\[
t_e = n - \hat{i} + \rho - 2R \left[ \frac{\rho-(n+i)}{n} \right] + t_s \quad \forall \ i \in \{1, \ldots, n\}
\]

Then:

\[
< t_e-t_s > = < n-\hat{i} + \rho - 2R \left[ \frac{\rho-(n+\hat{i})}{n} \right] + t_s - t_s >
\]

\[
= n + \rho - < \hat{i} > - 2 < R \left[ \frac{\rho-(n+\hat{i})}{n} \right] >
\]

Since \( \hat{i} \) has \( n \) unique values \( : 1, \ldots, n \), then \( < \hat{i} > = 1 + \frac{n-1}{2} = \frac{n+1}{2} . \)

Now \( < R \left[ \frac{\rho-(n+\hat{i})}{n} \right] > = \frac{1}{n} \sum_{i=1}^{n} R \left[ \frac{\rho-(n+\hat{i})}{n} \right] \)

\[
= \frac{n-1}{2} ,
\]
by Sub-Lemma 1 which states that
\[
\frac{1}{n} \sum_{i=1}^{n} R \left[ \frac{m-i}{n} \right] = \frac{n-1}{2} ,
\]
where we substitute \((\rho - n)\) for \(m\) and \(\hat{i}\) for \(i\).

Hence:
\[
< t_e - t_s > = n + \rho - \frac{n+1}{2} - 2 \cdot \frac{n-1}{2}
\]
\[= \rho + 1 - \frac{n+1}{2} .
\]

CASE II: \(\rho \leq 2n\). Now,
\[
t_e = t_e^1 = n + \hat{i} - \rho + t_s \quad \text{if} \quad \rho < n + \hat{i}, \text{and}
\]
\[
t_e = t_e^2 = n - \hat{i} + \rho - 2R \left[ \frac{\rho-(n+i)}{n} \right] + t_s \quad \text{otherwise}
\]
where \(\hat{i} = 1,...,n\).

Thus
\[
< t_e - t_s > = \frac{1}{n} \sum_{\hat{i}=1}^{n} (t_e - t_s) = \frac{1}{n} \sum_{\hat{i}=1}^{\rho-n} (t_e^1 - t_s) + \frac{1}{n} \sum_{\hat{i}=\rho-n+1}^{n} (t_e^2 - t_s)
\]
\[= \frac{1}{n} (2n-\rho)(n-\rho) + \frac{1}{n} \sum \hat{i}
\]
\[= \frac{1}{n} \sum_{\hat{i}=\rho-n+1}^{n} (t_e^1 - t_s)
\]
\[= \frac{1}{n} \sum_{\hat{i}=\rho-n+1}^{n} (t_{e \leftarrow \hat{i}} - t_s)
\]
\[= \frac{1}{n} \sum_{\hat{i}} \hat{i}
\]
\[= \frac{1}{n} \sum_{\hat{i}=\rho-n+1}^{n} \left[ \sum_{\hat{i}=1}^{\rho-n} \hat{i} - \sum_{\hat{i}=1}^{\rho-n} \hat{i} \right]
\]
\[= \frac{1}{n} \sum_{\hat{i}=\rho-n+1}^{n} \left[ \sum_{\hat{i}=1}^{\rho-n} \hat{i} \right]
\]
\[= \frac{2n + 2\rho n - \rho - \rho^2}{2n}
\]
Thus
\[ \frac{1}{n} \sum_{i=p-n+1}^{n} (t_{i}^{1} - t_{s}) = \frac{4n^2 - 6p^2 + 2n + 2p \cdot n - \rho \cdot \rho^2}{2n} \]
\[ = \frac{4n^2 - 4p^2 + 2n - \rho}{2n} \]

And:
\[ \frac{1}{n} \sum_{i=1}^{p-n} \left( t_{i}^{2} - t_{s} \right) = \frac{1}{n} \sum (n-i+p-2R[i \frac{p-n+i}{n}]) \]
\[ = \frac{1}{n} (p-n)(n+p) - \frac{1}{n} \sum i - \frac{2}{n} \sum R[i \frac{p-n+i}{n}] \]
\[ = \frac{1}{n} (p-n)(n+p) - \frac{1}{n} \sum i - \frac{2}{n} \sum \frac{p-n-1}{n} R \left[ \frac{k}{n} \right] \]

where \( k = p-(n+i) \).

Now \( n < \rho \leq 2n \Rightarrow 0 < p-n \leq n \)
\[ \Rightarrow p-n-1 \leq n-1 \]
\[ \Rightarrow k \leq n-1. \]

Hence \( \sum_{k=0}^{p-n-1} R \left[ \frac{k}{n} \right] = \sum_{k=0}^{p-n-1} k \) since \( R \left[ \frac{k}{n} \right] = k \) if \( 0 \leq k \leq n-1 \)

Thus:
\[ \frac{1}{n} \sum_{i=1}^{p-n} \left( t_{i}^{2} - t_{s} \right) = \frac{(p-n)(n+p)}{n} - \frac{1}{n} \sum_{i=1}^{p-n-1} i - \frac{2}{n} \sum_{k=0}^{p-n-1} \frac{k}{n} \]
\[ = \frac{(p-n)(n+p)}{n} - \frac{3}{n} \sum_{i=1}^{p-n-1} i - \frac{p-n}{n} \]
\[ = \frac{(p-n)(n+p-1)}{n} - \frac{3}{n} \frac{(p-n-1)(p-n)}{2} \]
\[ = \frac{6p-n^2 + 5n^2 - n}{2n} \]
Finally:
\[ < t_e - t_s > = \frac{6\rho n - \rho^2 + 5n^2 - n + 4n^2 - 4\rho n + \rho^2 + 2n - \rho}{2n} \]

\[ = \rho + 1 - \frac{n + 1}{2} \]

We see that in either Case I or Case II we have
\[ < \gamma_i > = < t_e - t_s > = \rho + 1 - \frac{n + 1}{2} \]

This is a remarkable result.

\[ < \gamma_o > : \] Since \( n \) packets are supplied every \( \rho \) time units, it follows immediately that all packets are processed in unique processors; that is, all \( n \) processors are in use. Now, an output packet gated into output register \( i \) at time \( t \) will require

\[ \gamma_o^i = n - i + 1 \]

time units to be clocked into the acceptor. Note there are no output loop delays (theorem 2), and the acceptor does not allow recirculation of output packets.

Hence
\[ < \gamma_o > = < \gamma_o(i) > \]

\[ = \frac{1}{n} \sum_{i=1}^{n} (n - i + 1) \]

\[ = \frac{n + 1}{2} \]

Finally:
\[ < p > = < \gamma_s > + < \gamma_i > + < \gamma_o > \]
\[
\frac{\rho}{n} + \rho + 1 - \frac{n+1}{2} + \frac{n+1}{2} \\
= \frac{\rho}{n} + \rho + 1 \\
= \rho (\rho + \frac{1}{n}) + 1 
\]

Since \(<P>\) is independent of \(j\), it is the same for all values of \(j\). That is, \(<P>\) is a valid average for all the tasks generated during the running of the system (omitting startup behavior).

**Lemma 4:** If \(\rho\) is odd, then at most one processor will initiate execution at every clock pulse.

**Proof:** Assume two processors begin execution at the same time, \(t\). Then

\[2i - 1 + j\rho = 2i' - 1 + j'\rho\]

where \(i' \neq i\).

Hence

\[2(i-i') = (j'-j)\rho.\]

**CASE I:** \(j' = j\). Then \(i = i'\), which is impossible.

**CASE II:** \(j' \neq j\). Let \(k = j' - j\).

Then:

\[i - i' = \frac{k\rho}{2}.\]

Since \(i - i'\) is integral and non-zero, then \(k\rho\) must be even. Since \(\rho\) is odd, then \(k\) must be even as well. Thus \(|k| \geq 2\), and \(|i - i'| \geq \rho\).

Since \(i, i' \in \{1, \ldots, \min(\rho,n)\}\) by lemma 2, then this is clearly impossible. Hence, at most one processor can initiate execution at any clock pulse.
Theorem 3: If $\rho'$ is odd and $\beta = \frac{1}{2}$ then no processor will ever wait to use the bus.

Proof: The proof is contained within the text on page 23.
REFERENCES


