DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING
COLLEGE OF ENGINEERING & TECHNOLOGY
OLD DOMINION UNIVERSITY
NORFOLK, VIRGINIA 23529

ATAMM ENHANCEMENT AND MULTIPROCESSING PERFORMANCE EVALUATION

By

John W. Stoughton, Principal Investigator

Final Report
For the period ended March 31, 1994

Prepared for
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

Under
Research Grant NCC1-136
Paul J. Hayes, Technical Monitor
ISD-Information Processing Technology Branch

Submitted by
Old Dominion University Research Foundation
P.O. Box 6369
Norfolk, VA 23508-0369

August 1994

EXECUTIVE SUMMARY

This report constitutes the year end report for 1993 and the final report for NASA Cooperative Agreement NCC1-136. The research described in this report involves extensions of the ATAMM implementation issues on several fronts. One area concerns implementation of cyclo-statically assigned processors in a truly distributed ATAMM Multicomputer Operating System (AMOS). Research is summarized to describe the modeling concepts, cyclo-static scheduling, distributed AMOS testbed design, and experimental results. Another area is concerned with investigating the sparse implementation of AMOS at the hardware level. Design, implementation and experimental achievements are subsequently described.

The research described in this report is the subject of two masters' theses. The cyclo-static scheduling work is reported by [ROY 93], and the hardware scale AMOS implementation is reported by [SASTRY 94]. This report is a condensation of the two research theses and alterations in the prose and technical presentations of the theses have been made for clarity and brevity. ATAMM model background and attendant cyclo-static scheduling issues are presented in Chapter One. A discussion of the design of the distributed AMOS is presented in Chapter Two and in Chapter Three, the distributed AMOS testbed and an example AMG experiment is discussed. In Chapter Four, design issues relating to a hardware ATAMM based testbed are discussed. A discussion of the implementation and experimental results are presented in Chapter Five.

The extension of ATAMM as a strategy for cyclo-static scheduling provides the basis for a truly distributed ATAMM Multicomputer Operating System (AMOS) [ROY 93]. To carry out the experimental validation of these concepts an ATAMM multicomputing testbed has been developed. The testbed consists of six PC/ATs networked using a 10 MBps peer-to-peer ethernet network. An example Algorithm Marked Graph (AMG) was demonstrated using cyclo-static, block cyclo-static or static scheduling policies. The execution of a graph bearing self loops, forwarded data tokens and control buffers was performed. This particular example also demonstrated the testbed's ability to instantiate nodes for different iterations.

The performance of each graph was evaluated by inspecting FDT data on the ATAMM Analysis Tool. Performances differed slightly from ideal behavior due to a communication overhead resulting from the file management system of MS-DOS/network software and single ethernet channel access.

The regular structure of the ATAMM model and the control organization for AMOS suggests the applicability for imbedded fine grain multiprocessor implementations. The research addresses the development of a low level language hardware based control structure for AMOS that employing data structures at the bit and byte level [SASTRY 94]. An ATAMM testbed is described using embedded firmware on 68HC11 microcontrollers. The control structure used for AMOS in this research is based upon a message passing model employing a contention free modified token ring physical layer. The testbed consists of centralized graph manager and three processors. Experiments on the evaluation of several data flow graphs are reported using ATAMM evaluation software.

TABLE OF CONTENTS

PAGE

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ABBREVIATION	DESCRIPTION
ADG	Algorithm Directed Graph.
ADAM	ATAMM Dataflow Multicomputer.
AGM	AMOS Graph Manager.
AMG	Algorithm Marked Graph.
AMOS	ATAMM Multicomputer Operating System.
ATAMM	Algorithm To Architecture Mapping Model.
CAMG	Centralized AMOS Graph Management.
CB	Length of Control Buffer on a Control Arc.
CMG	Computational Marked Graph.
CCr	Critical Circuit.
CP	Critical Path.
Ctrl.	Control.
D	"Done/Data" sub-node of the simplified NMG.
DAGM	Distributed AMOS Graph Management
F	"Fire" sub-node of the simplified NMG.
FDT	Fire, Data, Time events for the ATAMM Analysis Tool.
FGDF	Fine Grain Dataflow.
FU	Functional Unit.
i	Iteration number i.
IE	Input Buffer Empty.
IF	Input Buffer Full.
IPC	Inter-Processor Communication.
LAN	Local Area Network. The testbed uses a 10 MBps thin-ethernet with peer-to-peer (Novell NetWare Lite) software.
LGDF	Large Grain Dataflow.
MIMD	Multiple Input Multiple Data (parallel computers).
NMG	Node Marked Graph.
N	Number of AMG nodes.
NCO	Number of Control Tokens Output.
NDO	Number of Data Tokens Output.
N_{PE}	Number of Processing Elements required to execute an AMG.
NT	Intrinsic Node Execution Time.

NCO	Number of Control Tokens Output.
NDO	Number of Data Tokens Output.
N_{PE}	Number of Processing Elements required to execute an AMG.
NT	Intrinsic Node Execution Time.
PE	Processing Element.
PID	Processor Identification Number.
PR	Process Ready.
R	(i) The number of processors required to satisfy a specified TGP. (ii) "Resource" command used by a CAGM.
RAM-disk	Portions of RAM of a Personal Computer configured as Disks.
R_{max}	The number of processors required to execute an AMG with TBO = TBO_{LB}.
RTI	Real Time Interrupt
SA	Static Assignment
STRA	Strobe Line A
STRB	Strobe Line B
TBI	Time Between Inputs.
TBIO	Time Between Inputs and Outputs (computing time).
$TBIO_{LB}$	Lower Bound on TBIO.
TBO	Time Between Outputs (throughput).
TBO_{LB}	Lower Bound on TBO.
TBO_{min}	Minimum TBO due to overhead requirements.
TCE	Total Computing Effort.
T_{wait}	The total waiting time between nodes of a node-loop.
(X,Y,Z)	Connection Matrix entries of the form : (Iteration Increment on Data Arc, Number of Initial Data Tokens, Length of Control Buffer on Corresponding Control Arc).

CHAPTER ONE
CYCLO-STATIC SCHEDULING

1.1 Background

The problem domain addressed by ATAMM includes Large (coarse) Grain Data Flow (LGDF) applications that are deterministic in nature. Large grain dataflow problems are decomposed into macro blocks of code (instructions) that get executed whenever required data is available at input. Of special interest are deterministic, iterative LGDF algorithms whose block computation times are assumed to be constant.

A dataflow algorithm is represented as a directed graph in which nodes and arcs stand for instruction blocks and their data dependencies, respectively. Given such a decomposed dataflow algorithm, ATAMM uses a set of marked graph models to expose its control as well as data dependencies. Using Gantt chart representations of algorithm performance in steady state, the ATAMM model specifies measures for throughput and computing time that form the criteria for predicting performance based on the number of available computing resources.

ATAMM is manifested in software as the ATAMM Multicomputer Operating System, AMOS, which is implemented on real-time multicomputer architectures to achieve predictable, reliable and deadlock-free performance of LGDF computations. Earlier versions of AMOS (developed at the NASA Langley Research Center), include the Advanced Development Module (ADM), a four processor architecture (VHSIC) based on the Westinghouse MIL-STD-1750A instruction set processor. Another version of AMOS has been developed for the Generic VHSIC Spaceborne Computer, GVSC, a spaceborne four processor breadboard also based on 1750A VHSIC processors MIELKE90].

In these embodiments of ATAMM, AMOS code is executed on all processors in a multi-threaded fashion. Processor assignment is governed by a processor queue that allows only one processor to schedule a node for execution at a time. This scheduling is dynamic in nature and is based on the current state of execution of the algorithm. It is necessary to redundantly broadcast AMOS data structures and computed data across all processors of the system. The operation of AMOS in the ADM and GVSC relies on preserving the total graph view across every processor at all times. AMOS operation in these systems may be termed as centralized since only one processor at a time can supervise a node scheduling operation. The assignment policy of a centralized AMOS with a single queue requires processors to be homogeneous. However, a distributed heterogeneous processing environment can be created by establishing different queues for each class of computing resource in the system [JONES ofr SOM 93].

An alternative to the above is a static and deterministic scheduling approach that establishes a specific mapping of nodes to processors for every iteration of the dataflow algorithm. Motivation for this form of scheduling is derived from observing the periodic execution behavior in deterministic LGDF algorithms. Of interest are scheduling operations which may be distributed among processors such that processors concurrently participate in scheduling nodes for execution. In this sense, AMOS graph management operations such as node scheduling and processor assignment may be performed in a distributed manner.

1.2 The ATAMM Dataflow Paradigm

The ATAMM model provides the analytical means to integrate algorithm dataflow with the target architecture. It is based on timed marked graphs which are suitable for describing data and control flow events within a computational system.

A set of three marked graphs, the Algorithm Marked Graph (AMG), the

Node Marked Graph (NMG) and the Computational Marked Graph (CMG) constitute the main components of the ATAMM model [STOUGHTON88]. Given an algorithm decomposition, the Algorithm Directed Graph (ADG) is used to describe data dependencies. An example ADG is portrayed in Figure 1.1(a).

The AMG is a marked graph representation of the ADG. Circles on the graphs denote algorithm nodes ("chunks" of macro dataflow code). The edges of the AMG represent data dependencies between predecessor and successor nodes, while bullets on these edges represent the presence of data tokens. Squares represent sources and sinks, thereby providing data entry and output collection points.

An algorithm node is enabled for firing when it has data tokens on all its incoming data arcs. The node fires by encumbering all input tokens, delaying for some time interval and depositing one data token on each outgoing edge. The AMG fuses algorithm data dependencies with the those imposed by execution requirements for the ADG. For example the AMG for the ADG in Figure 1.1(a) is presented in Figure 1.1(b). An initial data token has been deposited on the data edges from node two to one and node three to two, thus satisfying the initial execution conditions for the graph.

The AMG thus represents task decomposition and data dependence between processes in an effective fashion. Each node of the AMG is modelled with respect to data and control activities that is the basis of the ATAMM model. The Node Marked Graph (NMG) describes node specific activities and data dependencies that need to be satisfied. The architecture is assumed to have global memory for storage of data associated with each AMG edge. The global memory is distributed or shared among the processing elements (PEs). Each PE also contains local memory for the storage of data and the code to execute any node of the AMG. A PE must read data from global memory into its local data container, process the data and write the data back to global memory for access by other Pes. An additional requirement is that global

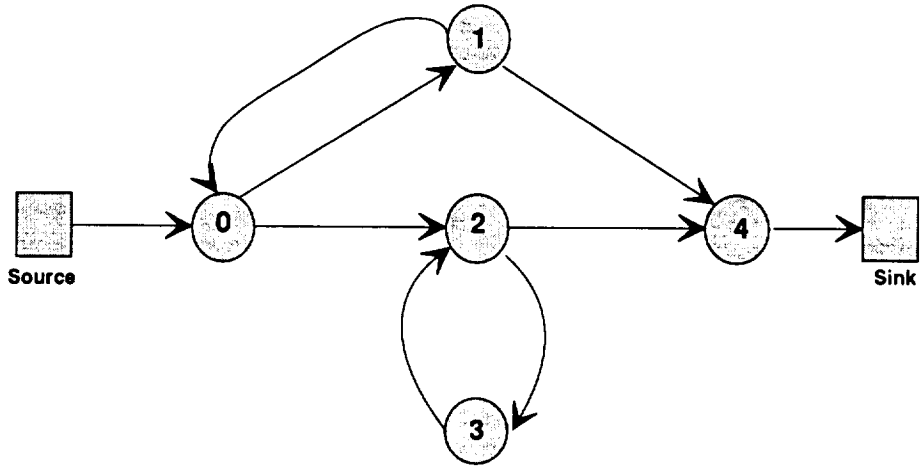


Figure 1.1(a) An Example Algorithm Directed Graph (ADG).

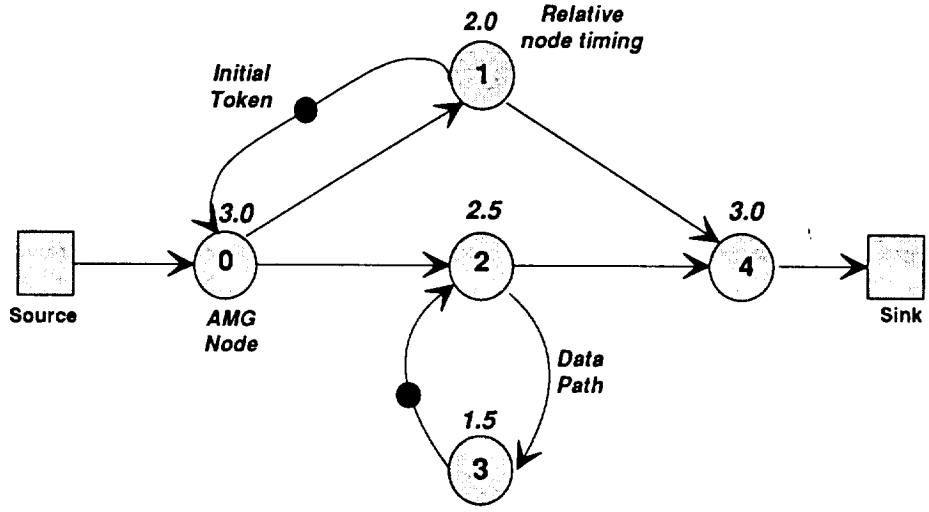


Figure 1.1(b) An Example Algorithm Marked Graph (AMG).

memory is available for output before node execution can commence. The NMG portrayed in Figure 1.2(a) describes the above node activities. The NMG specifies not only the activities to be performed at a PE but also the conditions which enable them to be performed. The read node cannot be fired on a PE until the processor is ready, input is available, and the output has been read by the successor operation. Once the PE is assigned to "fire" the read transition, it will remain assigned in order to process and write the data before becoming available once again [JONES90].

It can be shown that the NMG depicted in Figure 1.2(a) can be remodeled with fewer transition edges, which satisfy the necessary and sufficient conditions for algorithm execution [TYMCHYSHYN88]. The reduced NMG is presented in Figure 1.2(b). The reduced AMG contains "Fire" and "Done/Data" nodes as portrayed in Figure 1.2(c). The activities pertaining to the execution of a node are shown in Figures 1.2(d) and 1.2(e).

A fusion of the AMG and NMG marked graphs generates the Computational Marked Graph, (CMG). The CMG is constructed by replacing each AMG node is replaced with a NMG. The CMG built out of the AMG in Figure 1.1(b) is shown in Figure 1.3, with initial markings [JONES90].

The presence of CMG control and CMG data edges for every AMG data edge creates loops in the resultant CMG. Before a node in a dataflow graph can be fired, it must have a token on every incoming edge. Consequently, initial control tokens are needed on control edges, in order to ensure the first time execution of all AMG nodes. The initial tokens shown in Figure 1.3 satisfy initial control token dependencies of the graph. Note that data dependencies are automatically satisfied as the CMG is executed.

Special dataflow execution requirements (in terms of graph features) can also be incorporated in the CMG. For instance, in order to sustain a specific rate of execution, additional control tokens may be needed on incoming CMG control edges to a particular node. The presence of more than one control tokens on CMG control edges establishes CMG control buffers. Furthermore,

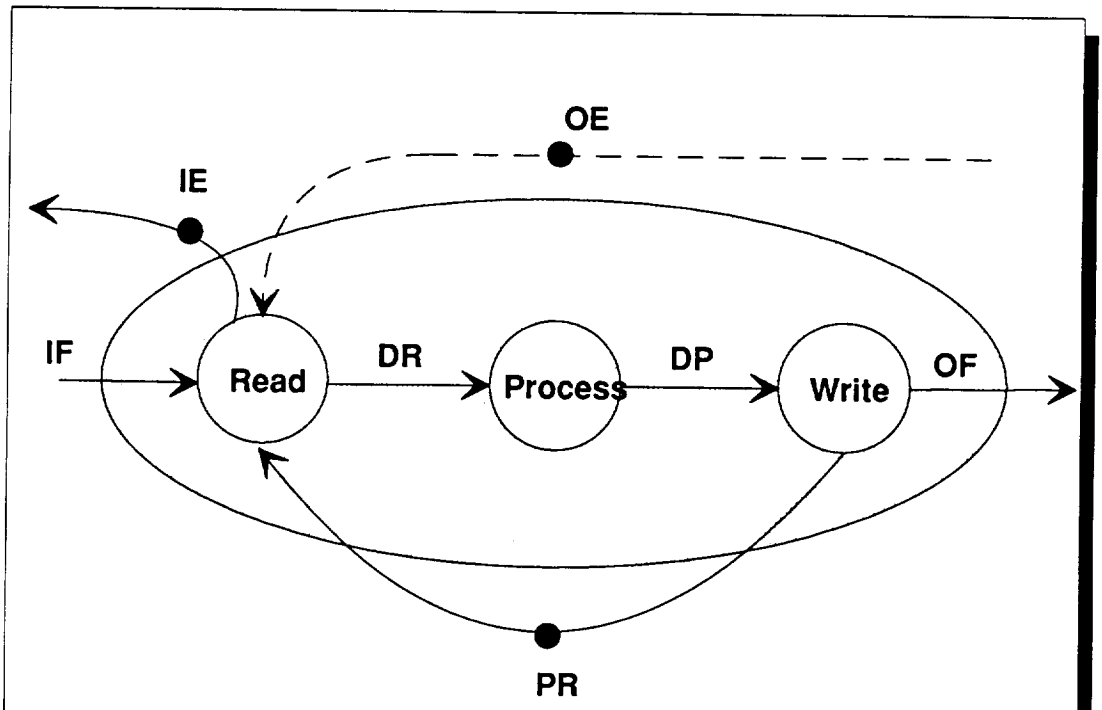


Figure 1.2 (a) Complex Node Marked Graph (NMG).

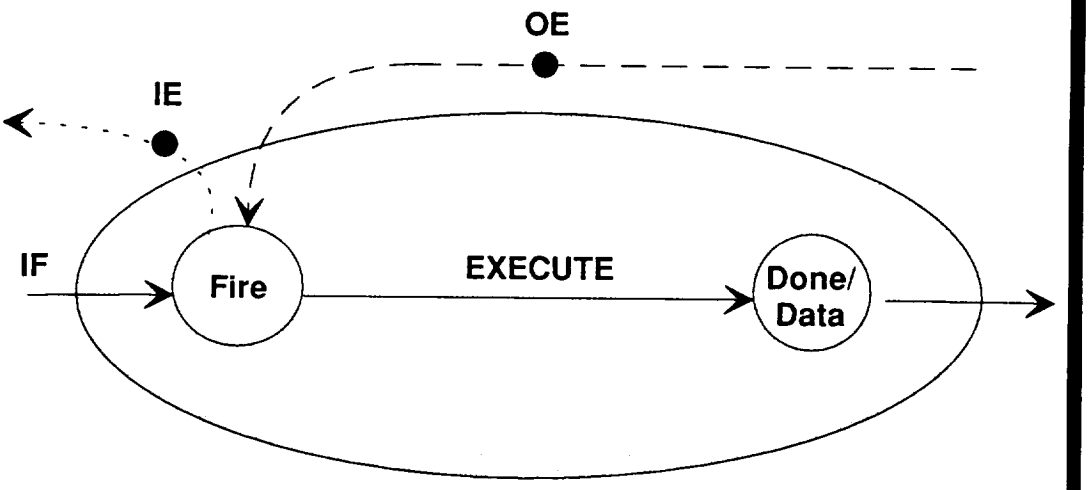


Figure 1.2 (b) Simplified Node Marked Graph (NMG).

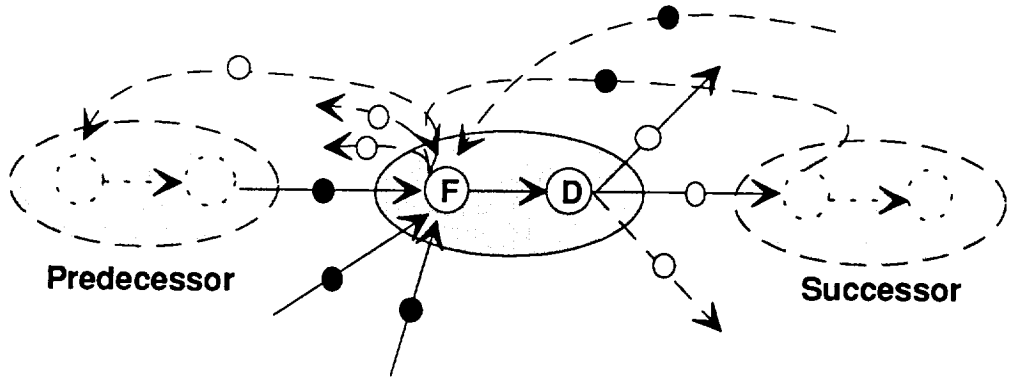


Figure 1.2 (c) Predecessor-Successor Node Relationships.

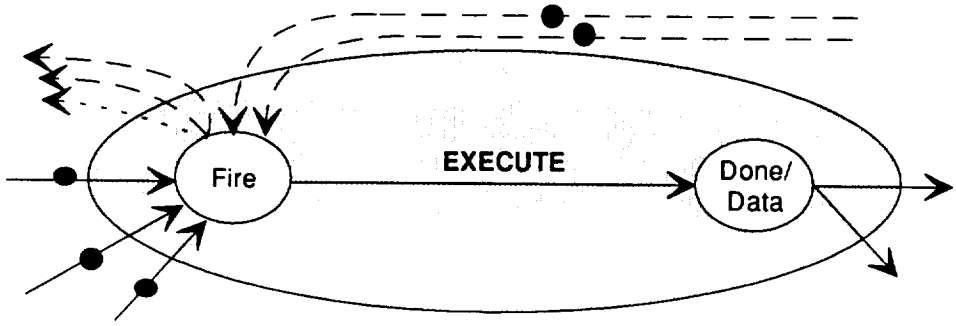


Figure 1.2 (d) Node Before Firing.

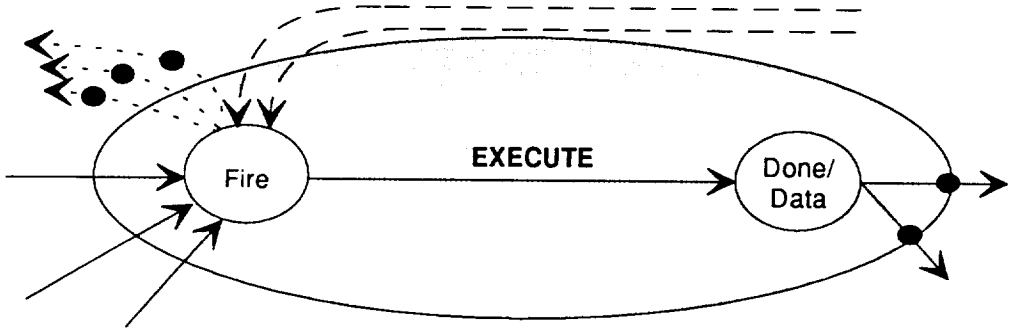


Figure 1.2 (e) Node After Firing.

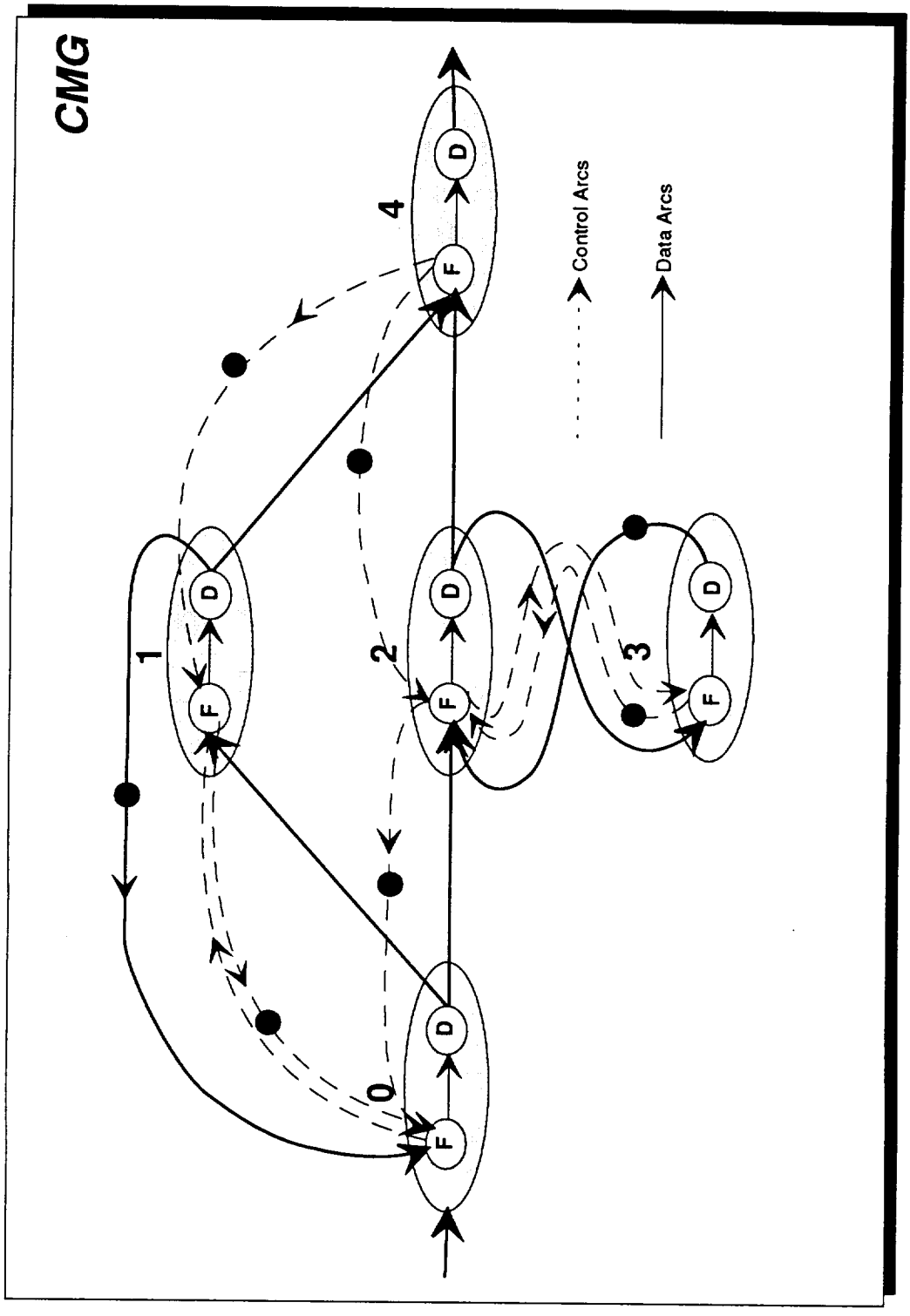


Figure 1.3 Computational Marked Graph for the AMG in Figure 1.1(b).

data tokens on outgoing data edges are usually generated for the current iteration index.However for certain types of data edges (such as loops), data tokens may need to be generated for future iterations. Such data tokens are termed as forwarded data tokens. Iteration increments corresponding to forwarded tokens can be marked against data edges in the CMG.

1.3 Performance Analysis for the ATAMM Modelling Process

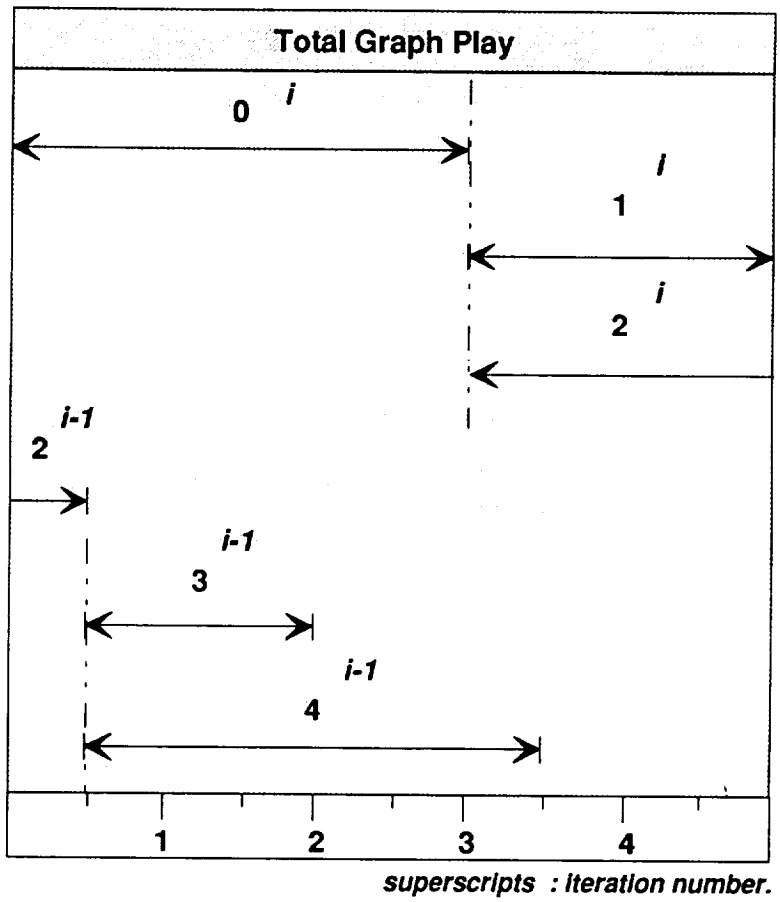
In the ATAMM environment, the execution patterns of a given CMG can be evaluated by using equivalent performance measurements, Time Between Outputs (TBO) and Time Between Input and Output (TBIO). TBO is a measure of the time interval between algorithm outputs and its inverse indicates throughput. TBIO is an indicator of computing latency [MIELKE 88] and [SOM 89].

The ATAMM model provides the means to match algorithm requirements with resource availability for achieving a balance between TBIO and TBO and also establishes the criteria for predictable performance. Predictability is attained by maintaining an input injection rate within the range determined by ATAMM [MIELKE90].

The stead state execution pattern may be viewed in a Gantt chard representation termed the TGP (Total Graph Play) diagram. For example, the TGP diagram for the AMG example of Figure 1.1(b) is shown in Figure 1.4.

The steady state resource requirement are obtained by counting the number of concurrently active nodes in the TGP diagram. The peak resource requirement, denoted by R_{max}, represents the maximum number of resources necessary to execute the graph with TBIO = $TBIO_{LB}$ and TBO = TBO_{LB}. However, for insufficient resources, performance cannot reach the bounds established by TBO_{LB} and $TBIO_{LB}$. Consequently, the resource requirement would be different from R_{max}, if an algorithm were to be operated with performance requirements other than $TBIO_{LB}$ and TBO_{LB}.

Performance metrics for algorithm execution are time measures that



TCE = 3.0 + 2.0 + 2.5 + 1.5 + 3.0 = 12.
TBO = 5.0.

Figure 1.4 TGP Diagram for the AMG in Figure 1.1(b).

characterize various aspects of run time behavior. A unit of computer time is defined by the product PE and one unit of execution time. For instance, the use of four PEs over ten units of execution time indicates 40 units of computer time. Computing Capacity, CC, is available computer time over an interval of time T. If R resources were available over T, the Computing Capacity is R * T units. Correspondingly, Computing Effort, CE, is defined as the units of computer time used over the interval T. Node Time (NT) is the time to execute a particular node. Total Node Time (TNT) is the sum of all the node times in an algorithm. The Total Computing Effort, TCE, is denoted by the total units of computing effort required by one processor to execute all AMG transitions once.

The relationship between TCE, R and TBO has been discussed by SOM, [SOM89]. That is, TBO for an algorithm marked graph operated periodically with R processors satisfies

$$[\text{TBO}] \geq [\text{TCE/R Processors}]. \qquad (1.1)$$

This can be restated as,

$$[\text{R Processors}] * [\text{TBO}] \geq \text{TCE} \qquad (1.2)$$

or

$$[\text{R Processors}] * [\text{TBO}] \geq [\text{TNT}] * [\text{1 Processor}] \qquad (1.3)$$

In other words, Computing Capacity (expressed in processor time units as R * TBO) equals or exceeds the Total Computing Effort exerted by a single processor. The expression [TNT] represents the aggregate time span that a single processor requires to discharge TCE units of "work".

1.4 Cyclo-Static Model

The steady state AMG execution, as characterized by the Total Graph Play diagram, is an instantiation of the AMG over a TBO time period. Within a TGP frame, each node of the AMG is executed once. Over M iterations, every AMG node is executed M times, giving rise to an ensemble of node execution traces. Of interest is the identification of one or more periodic node

sequences such that,

[1] each sequence represents a set of nodes containing exactly one instance of each of the N nodes of the AMG and therefore illustrating a single execution of every AMG node in a specific order;

[2] nodes are selected in a time exclusive manner so as to ensure that a node in the sequence is fired only after the completion of the node that precedes it in the sequence;

[3] an iteration index relationship gets established between successive nodes which ensures that while migrating from a node to an adjacent one, the iteration index gets incremented by zero or more; and

[4] the sequence is periodic across contiguous frames of length R * TBO. That is, if a node in the sequence is associated with iteration *i* in a given frame, it gets associated with iteration *i+M* in the next frame.

Due to its periodic behavior, a node sequence forms a *node loop*. The frame of length M * TBO, for which the node loop is identified, may be termed as a *loop frame*. Note that for any node loop beginning with node N_i in the first TBO frame, there exists another node also beginning with node Ni in the next TBO frame and so on for each of the TBO frames. This is to be expected since N_i is found in each of the TBO frames and thus the loop frame contains M node loops which are periodic and cyclically shifted by TBO. Note that each node loop identifies each node with a different index, and all nodes are selected in the loop frame. Thus a node loop set in the loop frame is mutually exclusive in node and index identification and collectively exhaustive in identifying all nodes.

1.4.1 Processor Requirements

Consider again a node loop over M*TBO. After completing a node, a processor may be required to wait to execute the next node in the schedule.

Over the entire schedule, the aggregate time a processor waits in order to be assigned is termed T_{wait}. Note that T_{wait} depends on the particular schedule. The sequence length per unit time satisfies

$$[M * TBO] = [TNT + T_{wait}] . \quad (1.4)$$

multiplying by a processor provides a relationship in units of computing effort for the execution of the AMG by one processor. Hence

$$[M * TBO]*1 \text{ Processor} = [TNT + T_{wait}]*1 \text{ Processor} . \quad (1.5)$$

Given that the AGM nodes are all executed in TBO time, then the number of processors required to perform the equivalent effort in TBO time, for the given schedule is obtained by dividing Equation 1.5 by TBO or

$$[M * \text{ Processors}] = \{[TNT + T_{wait}]/TBO\}*1 \text{ Processor}. \quad (1.6)$$

Thus, M processors are required per TBO time frame where $M \geq R_{max}$.

1.4.2 Cyclo-static node schedule

Assume the existence of periodic node loops that satisfy the criteria specified above. From the previous discussion R (R=M) processors are required for each TBO frame. Since each TBO frame consists of M mutually exclusive elements of node loops, then it is sufficient to map each of the R processors 1:1 to each of the node loops. The processor executes the node loop by migrating from one node to another in the prescribed sequence and by associating nodes with particular iteration indices. The processor repeats its cycle of execution periodically across consecutive loop frames. Consequently, it preserves a modulo R relationship between iterations associated with nodes. Each of the other R-1 processors also is assigned uniquely to one of R-1 node loops. Assigning a processor to a unique node loop guarantees that processors execute nodes in a mutually exclusive fashion over one node frame. These concepts are illustrated in Figure 1.5.

Furthermore, since every node that appears in a loop frame belongs to a particular node loop, the association of R processors with R node loops also establishes the collective exhaustiveness of the assignment process.

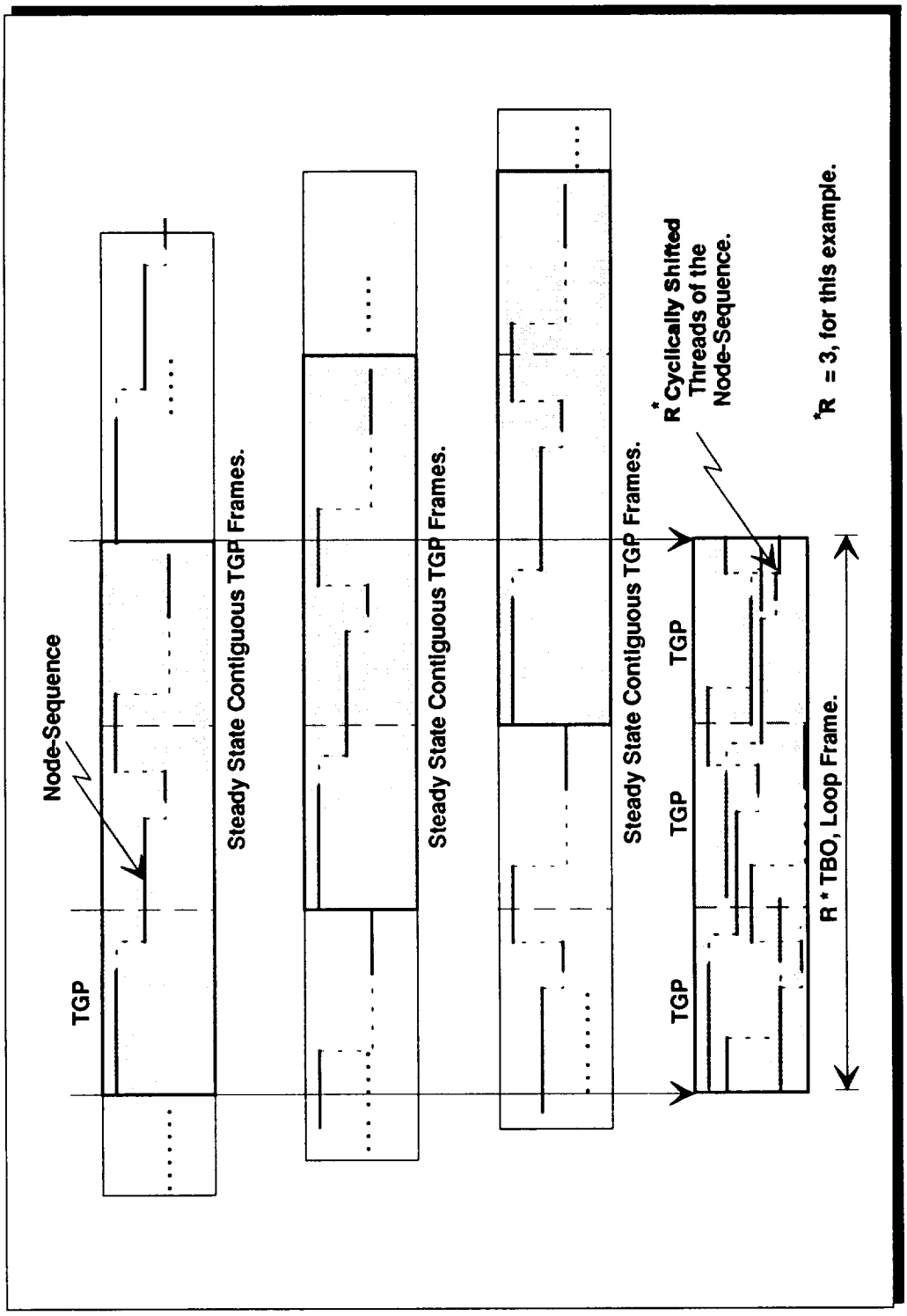


Figure 1.5 Existence of R Cyclically Shifted Threads of a Node--Sequence in the Loop-Frame.

The assignment described above is said to be fully *cyclo-static* and may be described more formally. If the schedule loop for processor R_k, $k \in \{0, R-1\}$, is periodic in R TGP frames, then processor R_k can be successively assigned to all nodes of the AMG, once, with a period of R * TBO. If resource R_k is assigned to node N_m, $m \in \{0, N-1\}$, in TGP frame k, $\forall k \in \{0, R-1\}$, then, resource R_j, $j \in \{0, R-1\}$, is assigned to node N_m in TGP frame $k \neq j$. Such a time interlocked relationship for every pair of resources $\{R_j, R_k\}$ and every node N_m, for one loopframe, establishes a scheduling loop for each resource in the system. Furthermore, if a node N_m is executed by processor R_k in iteration *i*, it is executed again by processor R_k in every iteration that satisfies a modulo(*i*,R) relationship.

A time measure for a cyclo-static assignment can be considered by reexamination of Equation 1.5. The processor assigned to a particular node loop produces TCE units of work since the execution of all N nodes of the AMG requires TCE effort. In addition, the term T_{wait} is the sum of all inter-nodal idle times present in the cyclo-static schedule loop and is dependent on the specific sequence selected.

It is to be noted that Equation 1.5 resolves the dependence between waiting time and the number of processors required. Given an AMG that requires a computing effort of TCE, a throughput of TBO, and a node sequence that contributes a inter nodal wait time of T_{wait}, R processors are necessary to execute the schedule loop among R processors in a cyclo-static fashion. Analogously, given TCE, TBO and a maximum resource availability of R, a cyclo-static schedule loop should be able to provide a T_{wait} that exactly satisfies Equation 1.5.

1.4.3 Block Cyclo-Static Scheduling

The inherent periodicity of node execution patterns in steady state, provides the potential for other types of cyclo-static behavior. Consider a scheduling policy using a set of schedule loops, each containing fewer than N nodes. In order to satisfy the parallel and pipeline concurrency present in

steady state, the set needs to contain two or more mutually exclusive blocks of limited node schedule loops. These blocks impose an implicit partitioning on the AMG nodes. Consequently, a given AMG node can appear in only one block in the set. Two or more processors could be assigned to periodically execute a particular block in a restricted cyclo-static manner. Similar assignments of processors to the remaining blocks of the system ensure the collective exhaustiveness of the assignment process. However, the processors now are scheduled to execute only those nodes that are contained in a block (as opposed to executing every AMG node in a cyclo-static schedule). Furthermore, the iteration numbers associated with nodes in a block bear a modulo R_b relationship, where R_b is the number of processors assigned to execute the block. With this scheduling policy, cyclo-static behavior is limited to executing blocks of schedule loops rather than a single N node schedule loop. Consequently, this form of cyclo-static behavior may be termed as *block cyclo-static*.

1.4.4 Static Scheduling

For a given AMG, if R blocks of schedule loops are created, each processor becomes solely responsible for a single schedule loop across all iterations. This represents an extreme form of block cyclo-static scheduling, which may be termed as *static scheduling*. The process is characterized by the division of an N node AMG into R mutually disjoint partitions. In particular, it should be noted that a fully static schedule consists of a set of R mutually exclusive node sequences, a block cyclo-static schedule contains k node sequences (where $1 < k < R$), and a cyclo-static schedule bears only 1 node-loop containing N nodes.

1.5 Schedule Loop Examples

Scheduling examples representing fully cyclo-static, block cyclo-static and fully static schedule loops are presented in Figures 1.6, 1.7 and 1.8, respectively. Schedule loops shown are with reference to the AMG of Figure

1.1(b), whose TGP appears in Figure 1.4. Every figure contains a "bubble diagram" representing a cyclical loop schedule for an assigned processor. Each node in a bubble diagram is associated with an iteration number which is unique with respect to the same node in any other bubble diagram. Note that the transition from a node to its neighbor may require an iteration number change. Furthermore, each thread of the schedule loop contains a node that is encapsulated by double lined circles. This is the initial node for the processor assigned to execute the thread. Determination of initial nodes is done by considering the steady state behavior of a node loop within a loop frame. Finally, a comprehensive list of all possible N node schedule loops for the example AMG is presented in Table 1.1 along with associated T_{wait} and R values. Each of these loops specifies a value of T_{wait} that requires the utilization of R_{max} processors. R_{max} is defined as the minimum number of processors required to execute an AMG in steady state for a given TGP.

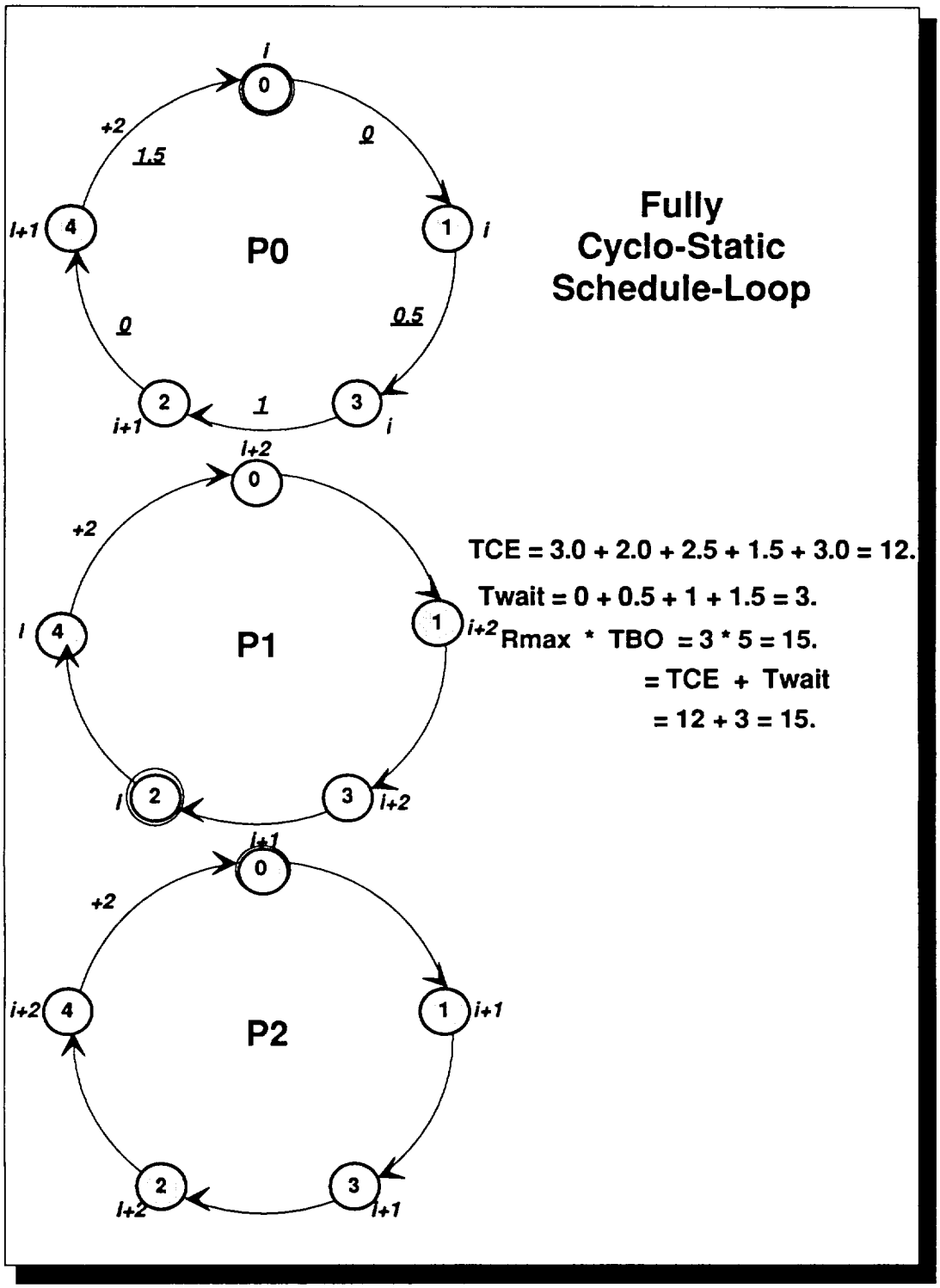


Figure 1.6 Fully Cyclo-Static Schedule-Loop for the AMG in Figure 1.1(b).

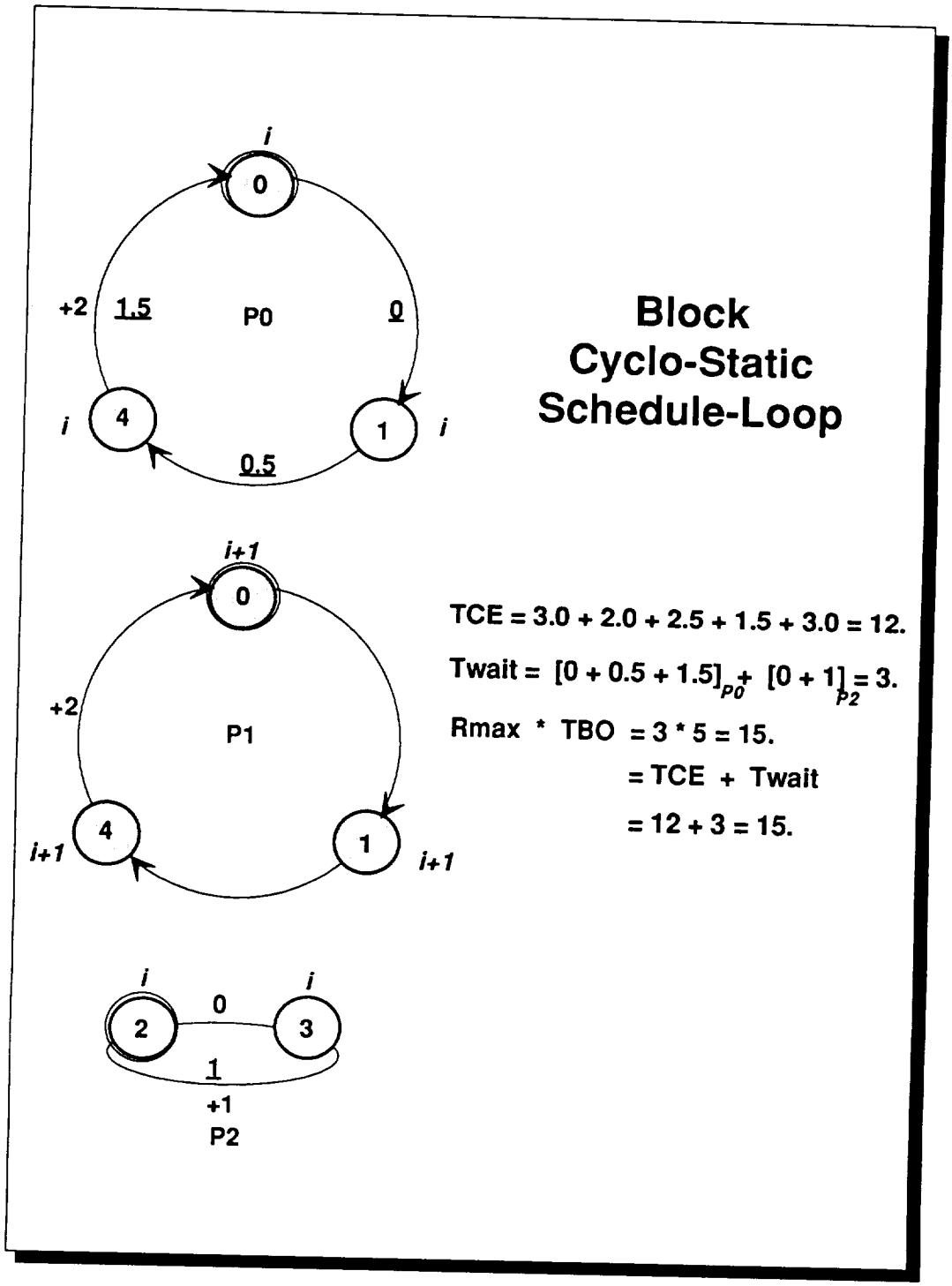


Figure 1.7 Block Cyclo-Static Schedule-Loop for the AMG in Figure 1.1(b).

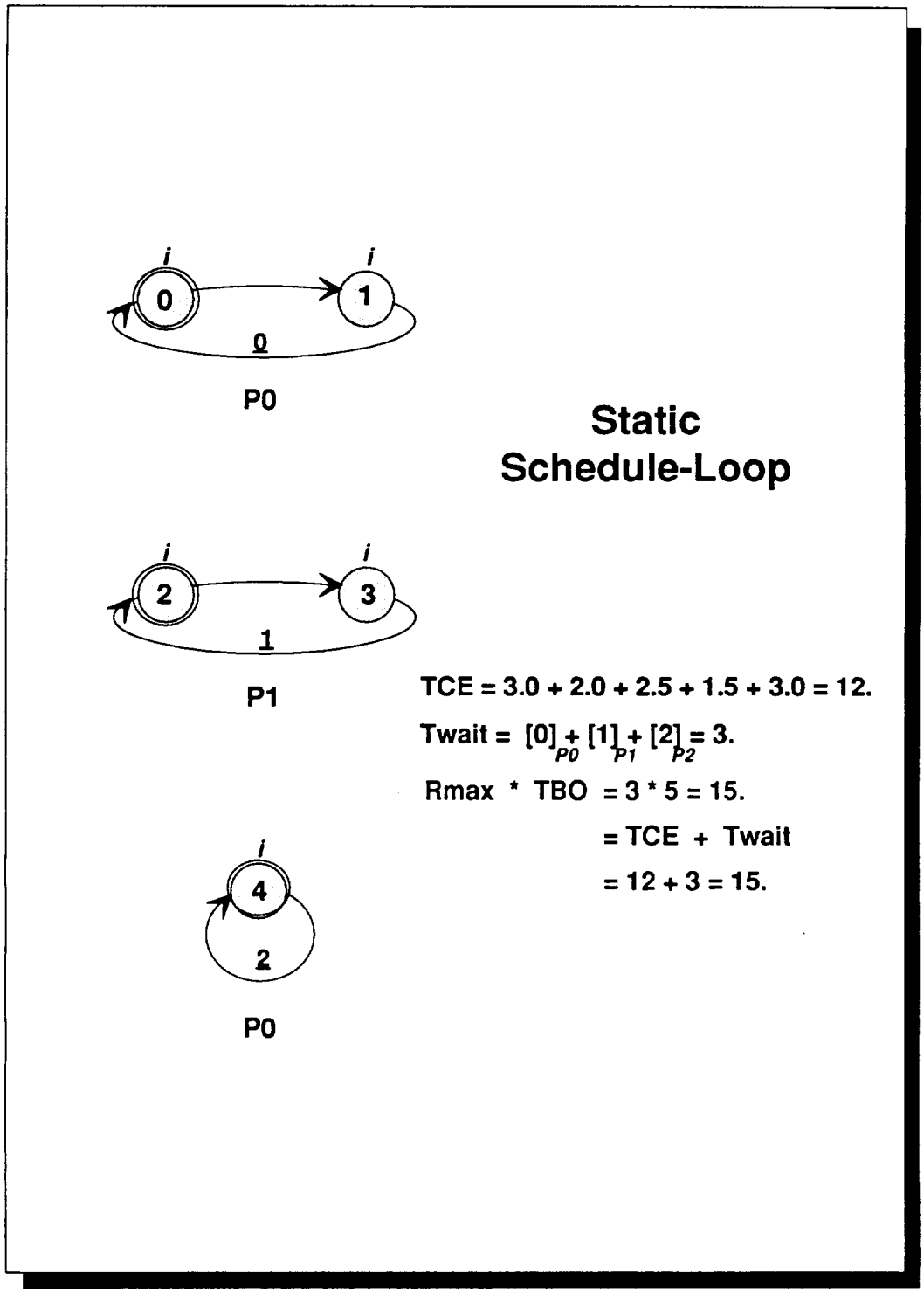


Figure 1.8 Static Schedule-Loop for the AMG in Figure 1.1(b).

Table 1.1 Cyclo-Static Schedule Loops for the AMG in Figure 1.1(b).

No.	Node Sequence	Processors	T_{wait}
1.	0 1 3 2 4	3	3
2.	0 2 3 1 4	3	3
3.	0 2 4 3 1	3	3
4.	0 1 2 3 4	4	8
5.	0 1 2 4 3	4	8
6.	0 1 4 2 3	4	8
7.	0 1 4 3 2	4	8
8.	0 2 1 3 4	4	8
9.	0 2 1 4 3	4	8
10.	0 2 4 1 3	4	8
11.	0 3 1 2 4	4	8
12.	0 3 2 1 4	4	8
13.	0 3 2 4 1	4	8
14.	0 4 2 3 1	4	8
15.	0 2 3 4 1	4	8
16.	0 4 3 2 1	4	8
17.	0 1 3 4 2	5	13
18.	0 3 1 4 2	5	13

Table 1.1 Cyclo-Static Schedule Loops for the AMG in Figure 1.1(b).			
19.	0 3 4 2 1	5	13
20.	0 4 1 2 3	5	13
21.	0 4 1 3 2	5	13
22.	0 4 3 1 2	5	13
23.	0 4 2 1 3	5	13
24.	0 3 4 1 2	6	18

TBO = TBO_{LB}. R_{max} processors are sufficient to satisfy the parallel and pipeline concurrency of an AMG.

CHAPTER TWO
DESIGN OF DISTRIBUTED AMOS GRAPH MANAGER

2.1 Distributed AMOS Overview

The ATAMM Multicomputer Operating System AMOS, is a logical interface between the dataflow graph and the multicomputer hardware. The basic elements of the ATAMM based computer is represented in Figure 2.1. The main components of AMOS include a variety of data structures which translate graph and execution parameters into an AMOS readable form, a graph manager, and a system task scheduler.

AMOS data structures represent the computational problem and the specific dataflow execution paradigm to be satisfied. Some of these structures are a graph connection matrix representing the AMG's data connections, information pertaining to buffer lengths and iteration increments along control and data arcs, initialization information etc.

The AMOS Graph Manager (AGM) performs the task scheduling operations of the operating system. The AGM may be classified as centralized or distributed, based on how CMG information is used during graph play. The taxonomy pertains to either having a single task master, which concerns itself with a composite view of the CMG at every point of execution, or having an ensemble of logically coherent AGM pieces which are responsible only for executing sub-partitions of the CMG. This essentially implies that a centralized graph manager (CAGM) exists as a monolithic task scheduler of all node activities as required to play the CMG. A Distributed AMOS Graph Manager (DAGM), on the other hand, is partitioned into unique but logically contiguous fragments. Each fragment monitors and executes a unique partition of the CMG. Collectively all fragments discharge the required scheduling of the AMG.

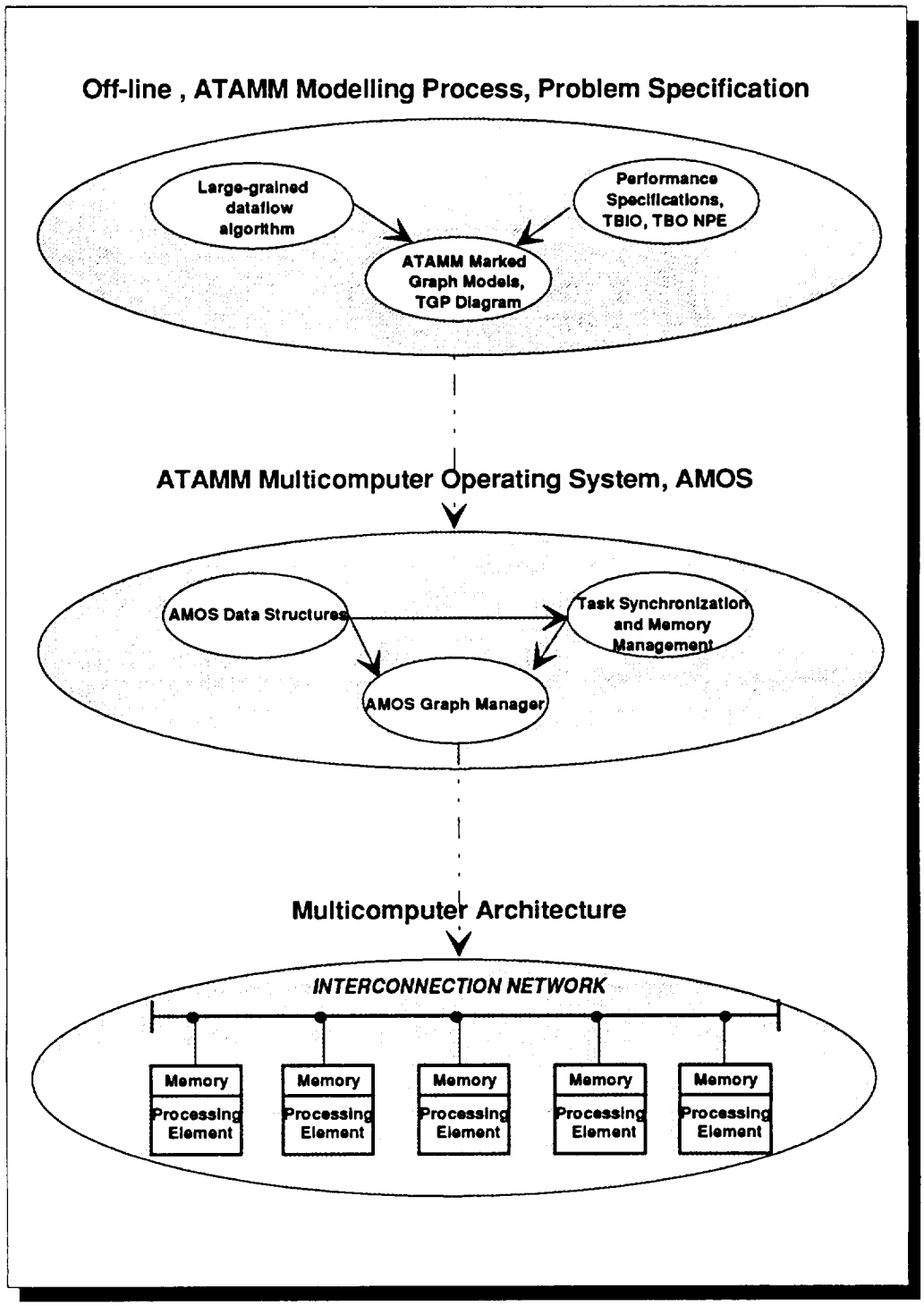


Figure 2.1 Key Elements of an ATAMM Dataflow Multicomputer.

2.1.1 Information Base for Distributed AMOS

The development of a distributed AMOS is based upon an information base that correlates processor assignment operations, cyclo-static node schedules and iteration index relationships. For a given AMG, a node loop specifies a sequence of nodes that need to be executed chronologically. It also indicates iteration increments associated with node transitions in the loop. Thus, each cyclically shifted node loop becomes preassigned to each of the R processors of the system. Attributes of this policy are that

[1] processors deterministically migrate from one node to another, in the manner established by the predetermined scheduling policy;

[2] reference iteration indices are updated while migrating from node to node as specified in the loop sequence; and

[3] each node loop is repeated periodically, thereby maintaining a unique modulo R relationship for the iteration indices associated with nodes in the loop.

The benefit of this schedule policy is that processors can sequentially schedule nodes for execution, without incurring any run-time scheduling overhead. The collective but autonomous execution of all node loop threads results in a conflict free execution of the AMG. The fact that a processor is made exclusively responsible for executing a sequence of nodes periodically for specific iterations provides a basis for a strategy for self governed and decentralized scheduling operations for an AMOS graph manager.

When assigned to a node loop, a processor remains in a state of continuous assignment, during which it picks up a node for execution, operates on the node, terminates its execution, and seeks to repeat the cycle on the next node in the schedule loop. However, this view of the assignment policy needs to be augmented with the inclusion of dataflow operations that satisfy the data and control dependencies of the underlying CMG. For example, before firing a node for iteration i, a processor has to transmit control

tokens to predecessor nodes for a future iteration $i+CB$ (where CB is the length of the control buffer for a given control arc). Analogously, after completing a node, a processor generates data tokens for successor nodes (which are associated with the present or future iterations).

Based on the above description of node scheduling and maintenance of graph control and data dependencies, the required pieces of information that suffice a strategy for self governed operations directly follow. Thus each processor needs:

[1] a view of the CMG which indicates data and control relationships between nodes;

[2] a schedule loop which specifies which node to do next;

[3] relative iteration indices for every node in the loop;

[4] a starting node associated with an initial iteration number to begin loop execution;

[5] information which details processor assignments for every node, for R successive iterations; and

[6] a modulo coefficient to associate with the repeated execution of each node in the schedule loop.

The first information element relates to a representation of the CMG that indicates data arcs (and therefore control arcs) and special AMG features such as control buffers and forwarded tokens relative to each node. In reference to the example AMG of Figure 2.2, Tables 2.1 and 2.2 portray the basic AMG connection matrix and its transpose, respectively. The features of the augmented AMG is shown in Table 2.3.

A schedule loop pertains to a specific sequence of nodes that every processor in the system executes. Since a processor executes a node for only a specific modulo R iteration number, the nodes of each thread that is assigned to a processor need to be tagged with relative iteration numbers. Threads for a schedule loop differ not only with respect to iteration indices for constituent nodes, but also in the node positions where assigned processors commence with

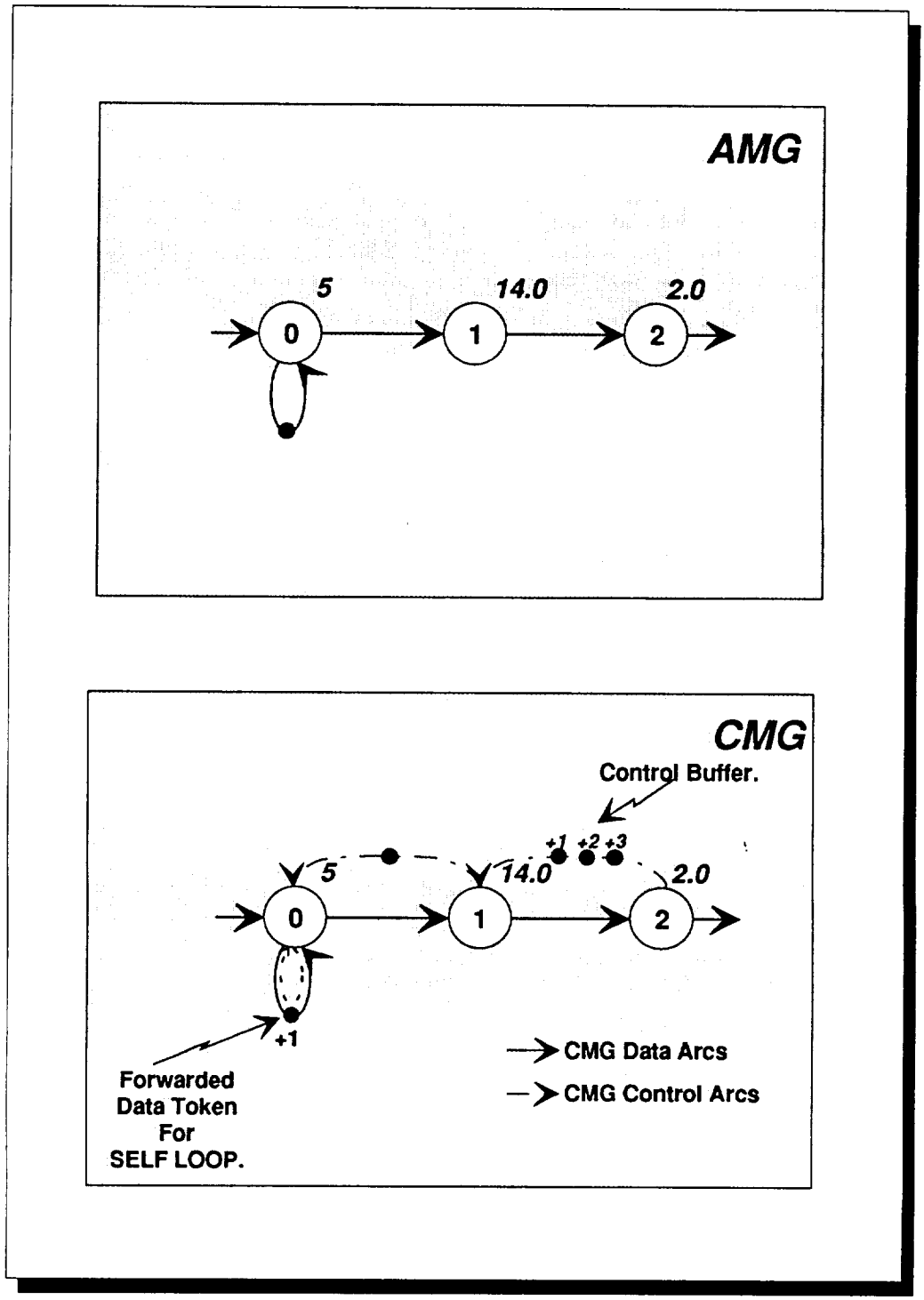


Figure 2.2 AMG That Requires Control Buffers and Forwarded Data Tokens.

Table 2.1 Basic Connection Matrix for the AMG in Figure 2.2.*

	0	1	2	3	4	5	6	7
0		**0**	**0**					
1	**0**				**0**			
2				**0**	**0**			
3			**0**					
4								
5								
6								
7								

*[ ] indicates the absence of an arc; 0 indicates an arc from row to column nodes.

Table 2.2 Transpose of Connection Matrix for the AMG in Figure 2.2.

	0	1	2	3	4	5	6	7
0		**0**						
1	**0**							
2	**0**			**0**				
3			**0**					
4		**0**	**0**					
5								
6								
7								

*[ ] indicates the absence of an arc; 0 indicates an arc from row to column nodes.

Table 2.3 Augmented Connection Matrix for the AMG in Figure 2.2.*

	0	1	2	3	4	5	6	7
0		**0,0,1**	**0,0,1**					
1	**1,1,0**				**0,0,1**			
2				**0,0,1**	**0,0,1**			
3			**1,1,0**					
4								
5								
6								
7								

* Each (x,y,z) entry indicates (data arc iteration increment, initial data tokens on data arcs, buffer length for control arc).

loop execution. Consequently, a processor needs to be assigned to a starting nodeand an initial iteration number for which it begins executing the node. A scheduling table for the example AMG is shown in Table 2.4. The initialization conditions for the example AMG is shown in Table 2.5.

Note that the execution of a node is accompanied by the generation of CMG control tokens for future iterations and of data tokens for present and/or future iterations. Accordingly, these tokens must be forwarded to processors which execute the nodes that actually require them. Due to the mutually exclusive but periodic execution of R unique node loop threads by R processors, it becomes feasible to foretell the processors that are assigned to execute a particular node for R successive iterations. Consequently, if a token is to be generated for a particular node for a specific iteration number, it is possible to target the token to the unique processor that will execute the node.It may be noted that this requirement is directly related to restricting unnecessary token movement via broadcast within a dataflow multicomputer.

For block cyclo-static or fully static scheduling policies, nodes belonging to a particular block are associated with iterations that bear a modulo R_b relationship, where R_b is the number of processor assigned to repeat the block in a cyclo-static manner. Consequently, the modulus coefficient R_b associated with every node for a block cyclo-static or static schedule needs to be specified. Observe that for cyclo-static schedules, all nodes possess a modulus coefficient that equals R.

In order to communicate data and control tokens to successor and predecessor nodes for specific iterations, a processor needs to know the processor assignments of these nodes for the required iterations. This information is relayed through an assignment table. For a given node and iteration number, the assignment table identifies the processor that executes the node. The example assignment table is shown in Table 2.6. The modulo-R index relationships are defined in a modulo operator table, which specifies the modulo number associated with every AMG node. This is shown in Table 2.7.

Table 2.4 Scheduling Table		
Pr. Node	Next Node	Iter.Incr.
0	1	0
1	0	+1
2	3	0
3	2	+1
4	4	+1

Table 2.5 Initialization Table		
PID	Init. Node	Init.Iter.
0	0	0
1	2	0
2	4	0

Table 2.6 Assignment Table

For a given node number and a modulo(iteration, operator) value, this table specifies the processor which shall execute the node.

Node #	0	1	2
0	0	0	0
1	0	0	0
2	1	1	1
3	1	1	1
4	2	2	2

Table 2.7 Modulo Operator Table

PID	% Operator
0	3
1	3
2	3

2.1.2 "FIRE", "EXECUTE" and "DONE/DATA" States

The availability of data tokens (from preceding nodes) and control tokens (from succeeding nodes) is the sole requirement that needs to be satisfied for enabling a node for execution. Once a node is found to be enabled, it informs its predecessor nodes of its "fire commencement" status, by transmitting control tokens to these nodes. Information about CMG data and control dependencies is derived from the connection matrix. In a multicomputer system, this step translates into three sub tasks:

[1] forming a data or control token entity with appropriate source, destination and iteration number;

[2] determining processors which are assigned to execute the predecessor nodes for particular a value of a future iteration and

[3] invoking AMOS IPC functions to physically transmit these control tokens to appropriate destinations (processors).The functions described above are subsumed within the "FIRE" state in the AMOS state diagram as shown in Figure 2.3. After executing a node, a node is ready to transmit computed data. It repeats the three steps, with respect to transmitting data tokens to successor nodes (after identifying recipients and building appropriate data token headers). These sub tasks are outlined in Figure 2.4.

With the termination of data communication, the overall execution of the current node is concluded. Using a cyclo-static scheduling policy (AMOS data tables), a processor determines a node that it can execute next. This function satisfies the second design criterion of ensuring self determined scheduling policies. Combining the operations of firing, execution, communication and task scheduling results in a state machine view for a distributed AMOS, which is presented in Figure 2.5. A pictorial description of the interaction between AMOS states and the data structures identified in the preceding Section is portrayed in Figure 2.6.

2.1.3 AMOS Integration

The integration of the AMOS components as generic Multicomputer Operating System (MOS) features, lends structure to the AMOS state machine. Briefly, the key components of a MOS are:

[1] an *initialization and synchronization* mechanism that ensures orderly, lock stepped execution of all system operations;

[2] a *memory manager* that manages the usage of local and global memory and ensures code and dataset protection;

[3] a *communication manager* to handle inter processor communication, IPC, between multicomputer Pes;

[4] a *task scheduler* that schedules tasks that are ready for execution with available processors in a manner that prevents deadlocks and avoids abnormal program termination; and

[5] a *resource manager* that allocates(assigns), removes and manages computing resources (processors) within the system;

These functional constituents have been collapsed into three principal tasks, as shown in Figure 2.7. Correspondingly, the AMOS state machine has been molded to concur with these task descriptions.

2.2 LAN Based Communication Layer

A testbed design objective is to ensure simple inter-connections between processing elements and to ensure a modularity of hardware interfaces (which allows scaling). The intent is to efficiently integrate an OS level communication layer and the distributed graph manager requirements of AMOS. Consequently, one suitable interconnection mechanism for an ATAMM based system is a bus oriented environment. An easy method of achieving this across discrete personal computers is to network them through an ethernet LAN. The corresponding realization of the testbed hardware is portrayed in Figure 2.8.

Commercial LANs are configured to transfer files efficiently. Consequently, AMOS message passing was performed using the file transfer capabilities of MS DOS and network software. However, the overhead of

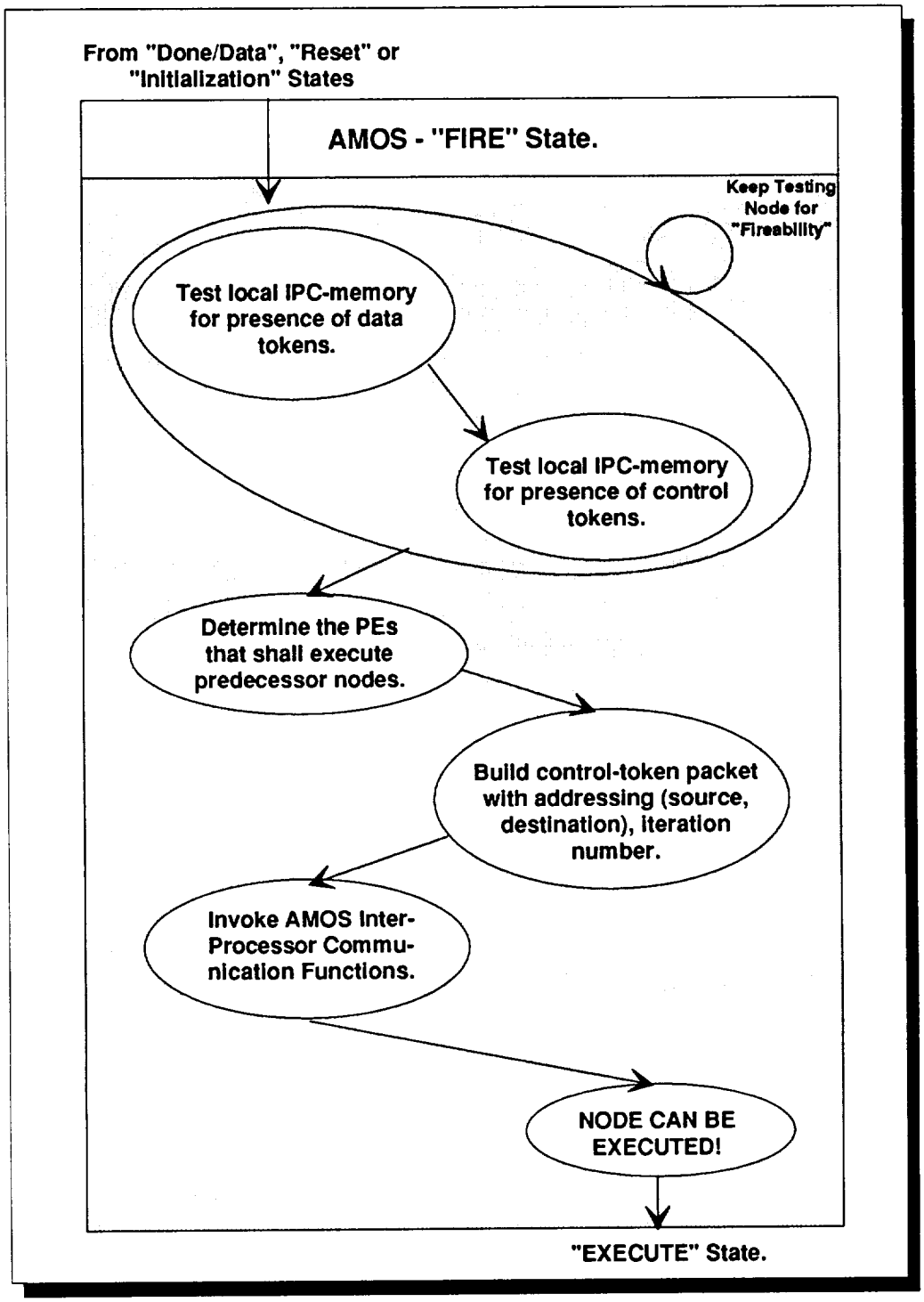


Figure 2.3 Events that Occur in the AMOS "FIRE" State.

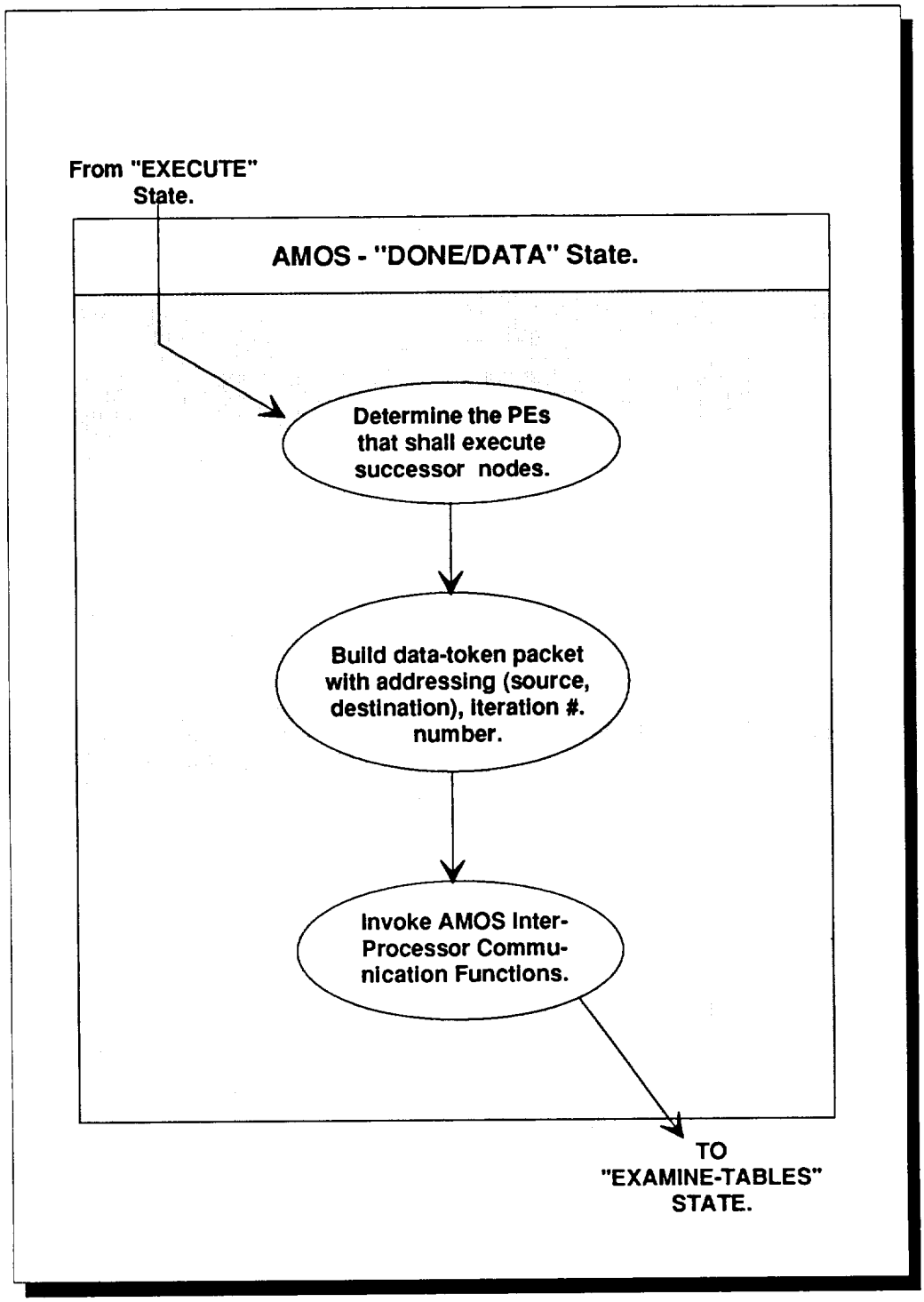


Figure 2.4 Events that Occur in the AMOS "DONE/DATA" State.

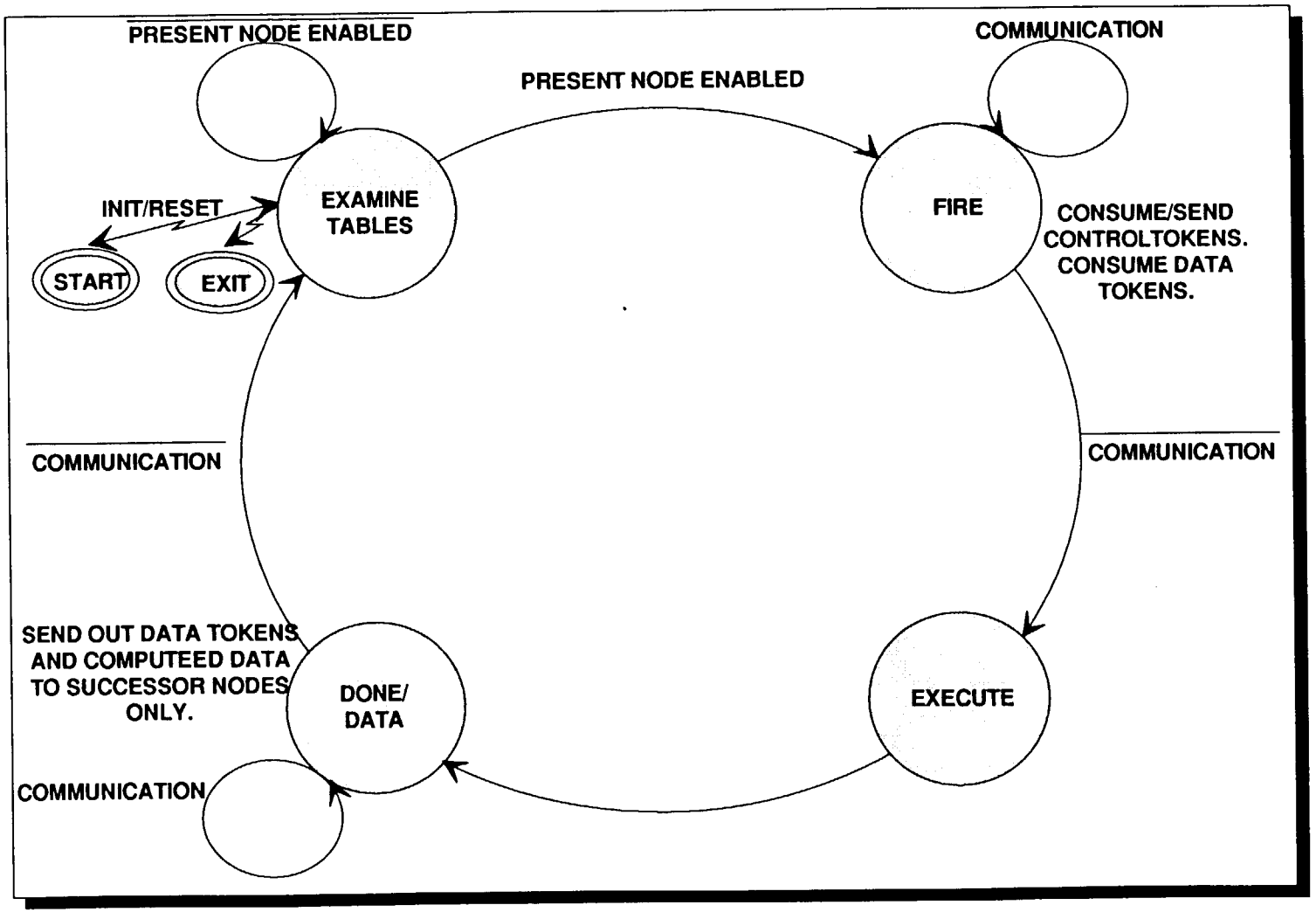


Figure 2.5 State-Machine View of Distributed AMOS Graph Manager.

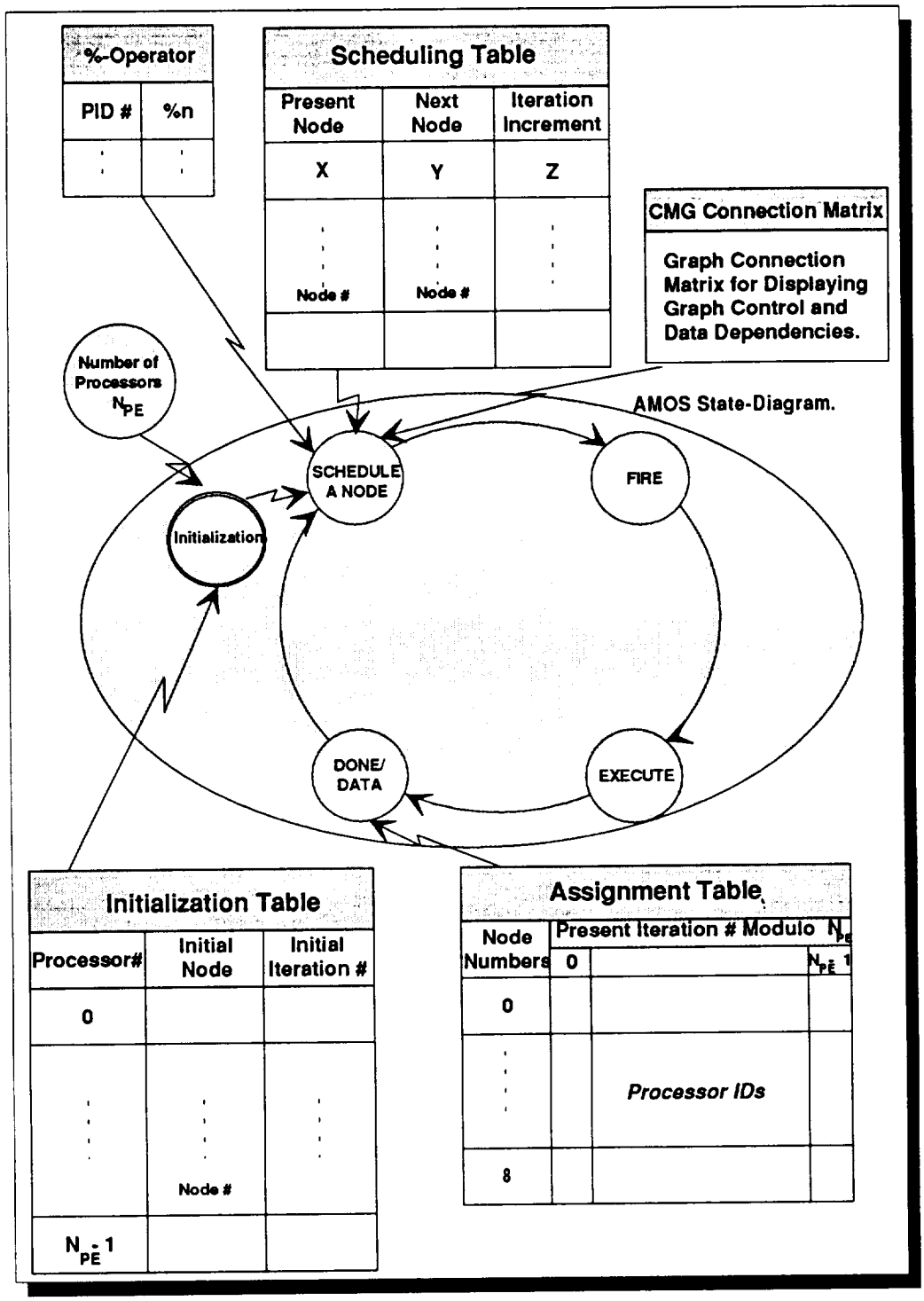


Figure 2.6 Interaction Between AMOS States and Data Structures.

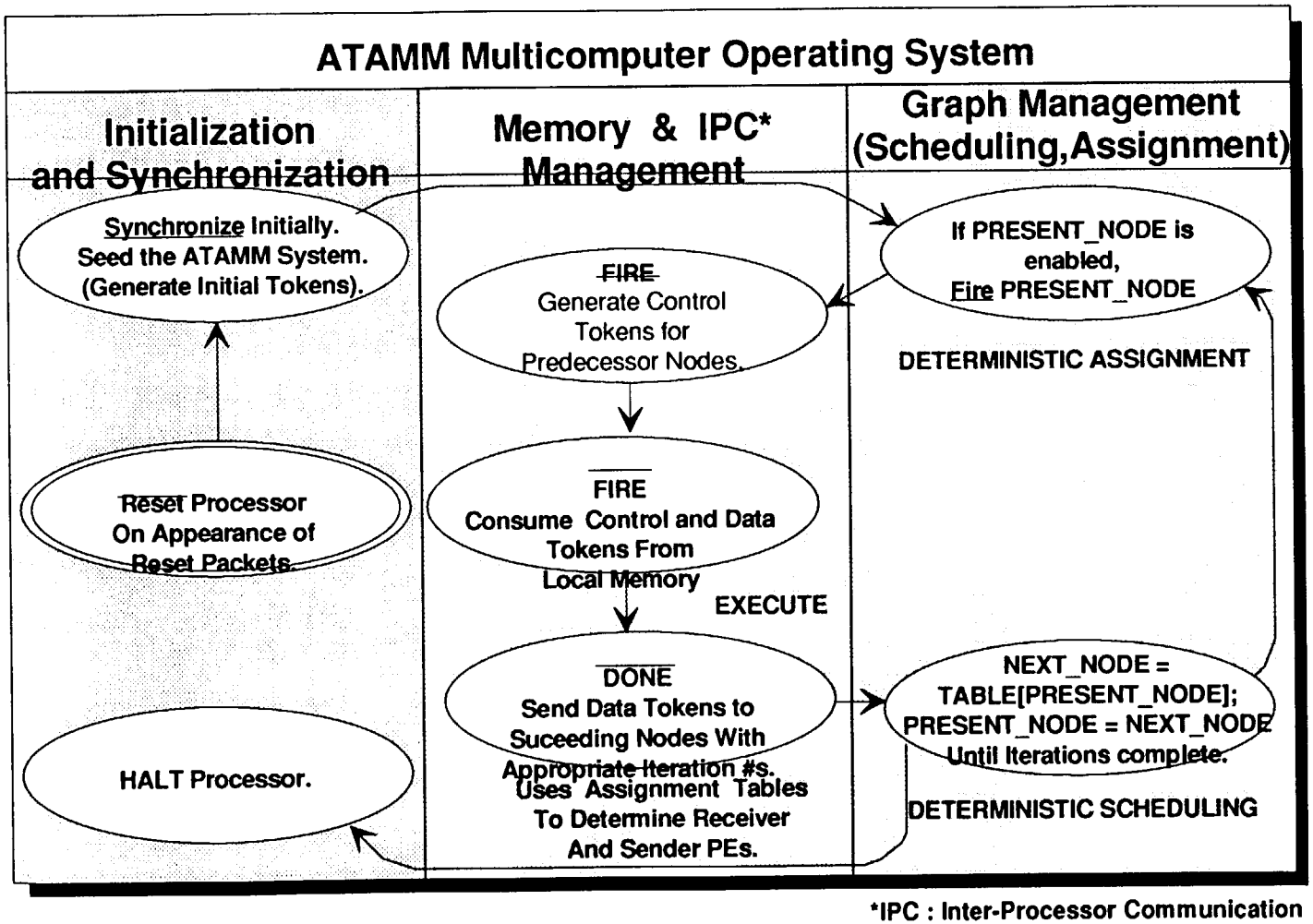


Figure 2.7 Distributed AMOS and its Functional Relationship with Standard Multicomputer Operating System Components.

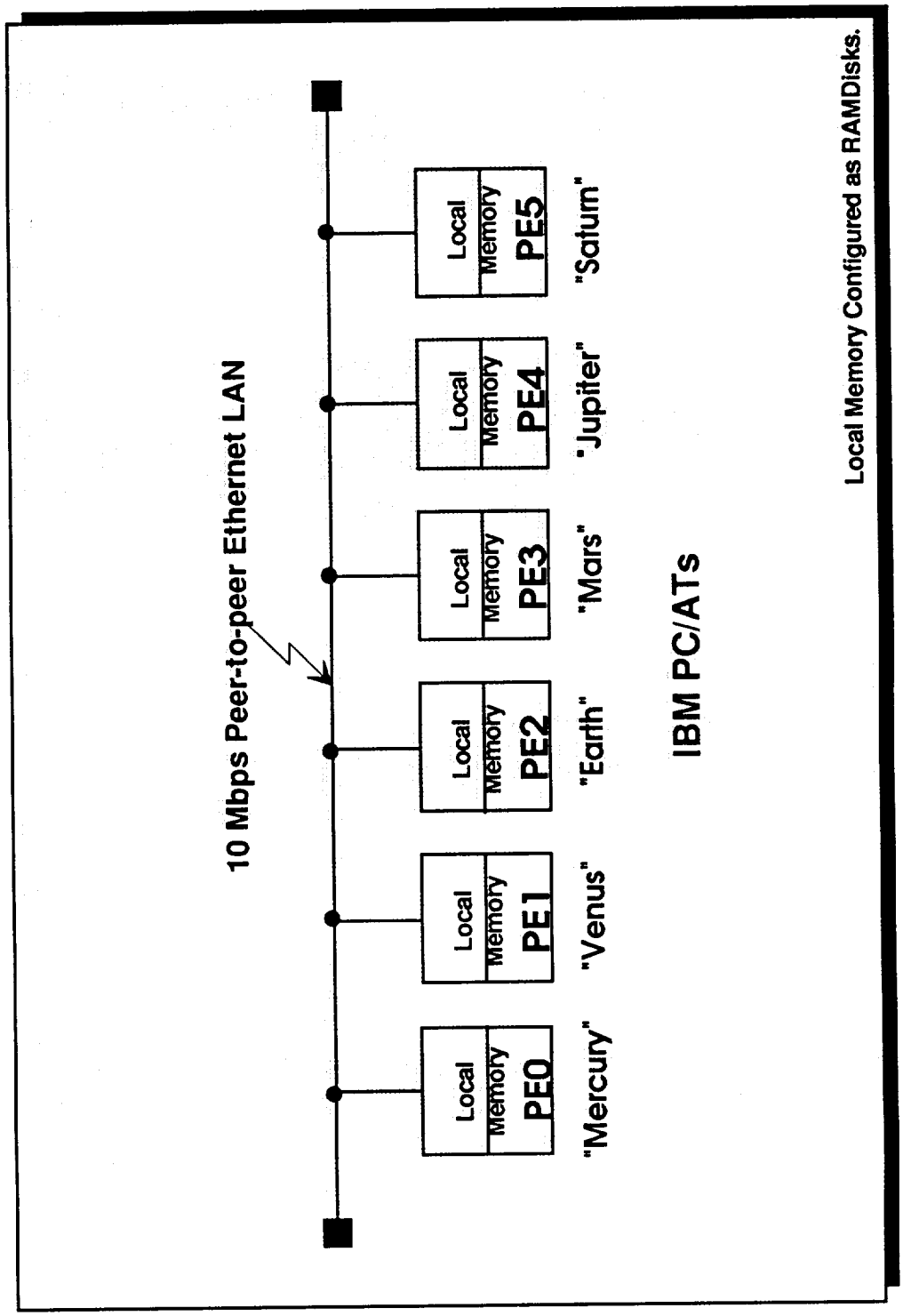


Figure 2.8 ATAMM Testbed Components : Processing Elements and Local-area-network.

file management is significant when one considers requisite file names, file handles, File Allocation Table, FAT entries, directory entries and other system data structures to be created and manipulated. Overhead time is increased further by the use of hard drives for file storage. In order to circumvent the serious communication delays that a true disk based file system would cause, RAM disks are used in place of hard disks.

Processors in a multicomputing system require local memory for storing dataflow algorithm code/data and operating system code/data. The testbed uses the conventional RAM as local memories for individual dataflow processors. In addition, a multicomputing system needs to possess some amount of shared memory which is accessible by all processors, in order to achieve message passing. One way of building a shared memory is to map portions of local memory on individual processors onto a shared memory space, such that every processor is granted access to these specially allocated local memories. Shared memory configured in the above manner is termed as Distributed Shared Memory, DSM, (an introductory discussion of which appears in [TANENBAUM92]). Portions of a DSM which form a logically contiguous, universal memory element, are actually distributed among processors of the system.

DSM can be modelled in networked environments by a LAN based system that offers a peer-to-peer operation, thus permitting individual computers to access directories on other computers. In our implementation, this translates into the capability of being able to access the RAM disk of every other computer. Consequently the operations of token movements translate into generating files that physically reside in the RAM disk spaces of destination processors. Correspondingly, in order to test for the presence of requisite data and control tokens (before firing a node), a processor only needs to look into its RAM disk directory for the availability of these tokens (files). Thus, local memory and memory for IPC are generated from a combination of conventional RAM and memory configured as virtual disks. In sum, the RAM

disk implementation integrates the concepts of memory communication and interprocessor communication. A description of the logical synapses between testbed components and the modelling of a DSM space is presented in Figure 2.9.

A summary of comparisons between the features of a centralized and distributed AMOS is presented in Table 2.8.

2.3 Testbed Operational Features

Features of the testbed include:

[1] The six processors of the testbed execute deterministic LGDF AMGs containing up to eight nodes. Prior to execution, a specific node-sequence needs to be identified for maintaining a desired TBO (throughput) and translated to fit the information structure of the testbed AMOS. Beyond this initial effort, testbed operation is autonomous and independent of any supervisory control.

[2] Execution of the AMG results in natural dataflow operations with highly diminished scheduling overhead (as compared to that incurred during dynamic scheduling).

[3] A distributed execution of AMOS graph management strategies is seen. Analogously, the execution of AMG nodes is also distributed.

[4] Communication of data and control tokens is performed on a need basis, meaning that only peer processors get involved in sending and receiving tokens.

[5] A degree of predictability is added to the ATAMM system since it can be predicted beforehand, which processor shall execute a given node for a particular iteration.

[6] Block cyclo-static or fully static schedules create the opportunity for executing LGDF algorithms in heterogeneous architectures.

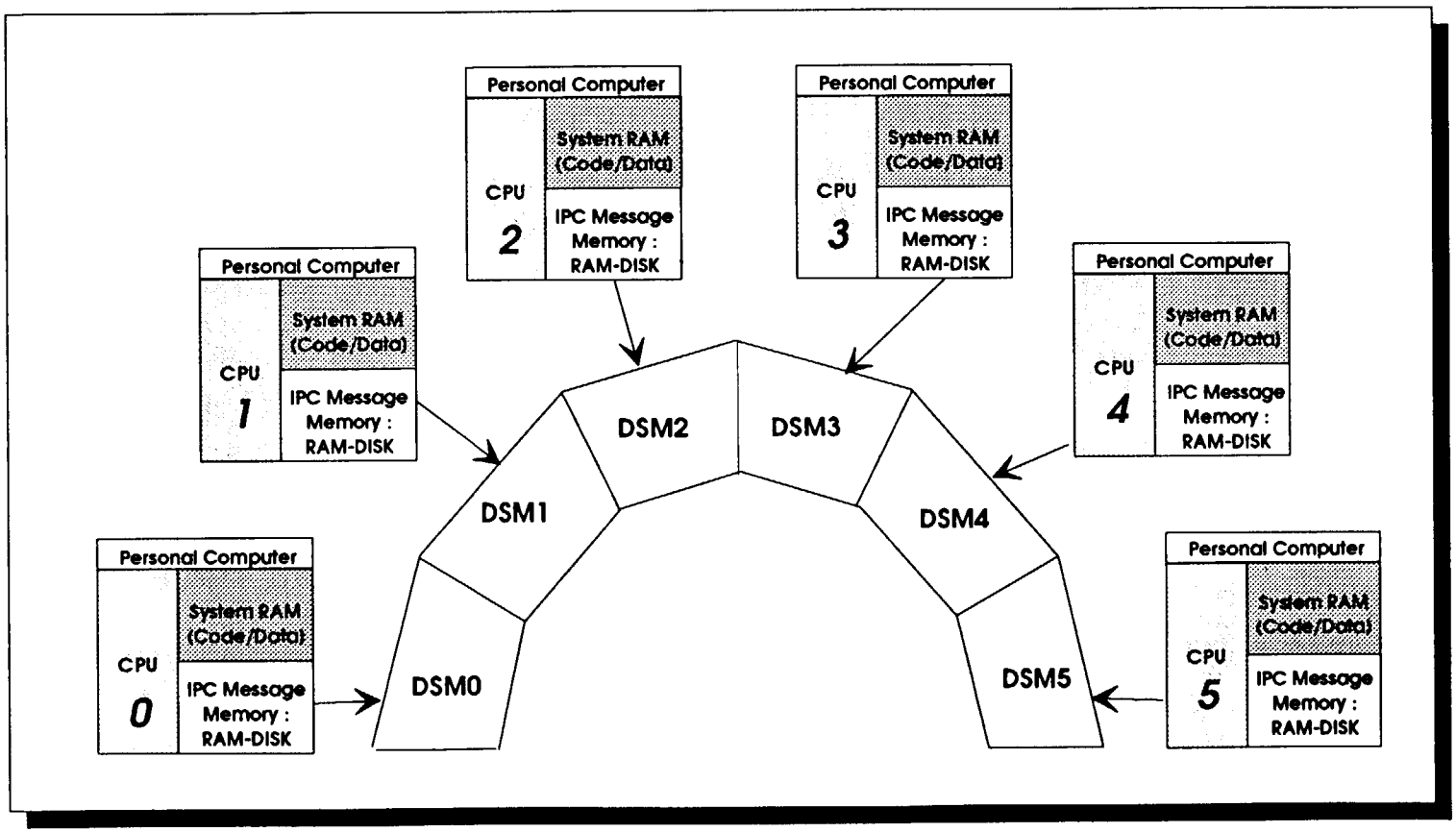


Figure 2.9 Distributed Shared Memory, DSM, for the ATAMM Dataflow Testbed.

Table 2.8 Features of Distributed and Centralized AMOS

#	Centralized AMOS	Distributed AMOS
1	At a given instant, only one processor can look for a schedulable node. Scheduling operations are considered to be centralized in this sense.	More than one processor can schedule a node for execution. Therefore, scheduling operations are considered to be distributed among numerous processors.
2	Processors have to be explicitly assigned for execution via means of a Processor Queue.	Processors remain in a state of continuous assignment.
3	AMOS is multi threaded.	AMOS is truly distributed.
4	Scheduling is dynamic, thereby incurring the overhead of run-time scheduling.	Scheduling is static since it is pre determined (during compile-time).
5	Scheduling (i.e. a mapping of a node to a processor) is non deterministic and unpredictable.	The processor assignment for a given node and iteration can be explicitly determined. Scheduling is highly deterministic and predictable.
6	Non deterministic scheduling requires a redundant broadcast of messages.	Deterministic scheduling allows specific message passing operations to be performed between peer processors.
7	Message passing involves F, D and R broadcasts, from which implicit token information has to be extracted.	Atomic token passing operations obviate the need to specially interpret messages.
8	Assignment and scheduling operations are disjoint. A processor is assigned when its PID surfaces to the top of the queue. Scheduling depends on the current state of the CMG.	Every processor remains continuously assigned for node execution and schedules a node by inspecting system tables that aid cyclo-static scheduling.

Table 2.8 Features of Distributed and Centralized AMOS		
#	Centralized AMOS	Distributed AMOS
9	Graph partitions cannot be handled naturally.	Block cyclo-static or static schedule loops allows graph partitions to be handle.

CHAPTER THREE
TESTBED EVALUATION

3.1 User Interaction

The testbed has been designed for autonomous operation that requires minimal user intervention. The user supplies AMOS node code and data structures such as the graph connection matrix, node loop and an initialization sequence. Problem specifications are listed out in a text file, which is subsequently distributed among the processors of the system. To suit certain aspects of experimentation, additional fields are introduced in the specifications file. For example, information pertaining to artificial node execution timings appear as node delays in current versions of the specifications file. A sample format of the specifications file appears in Table 3.1. For obtaining results from experiments, methods for statistical collection of results are necessary. The testbed reports results in three formats:

[1] Visual Display (of execution characteristics, global time etc.),

[2] A log of system activities in a text file,

[3] An FDT file in the ATAMM Analysis Tool format [JONES90].

A sample format appears in Table 3.2.

An experiment is begun with an AMG, a schedule, the subsequent formation of a specifications file and execution. Event timing is recorded in an FDT file for subsequent evaluation on the ATAMM Analysis Tool.

3.2 Testbed and Scheduling Evaluation

For brevity, a representation of numerous scheduling exercises on the testbed is presented. The demonstration AMG is that as shown in Figure 1.1(b) with associated TGP as shown in Figure 1.4. Featured is a

Table 3.1 Format of the Specifications File

Entry #	Entry Name	Description
1	Connection-Matrix for CMG.	[8 x 8 x 3] Matrix to indicate graph data structures. A non negative entry in the matrix indicates the presence of an data edge from row (predecessor node) to column (successor node). Each (x,y,z) entry indicates that x is the **iteration increment** for the corresponding data arc, y is the number of **initial data tokens** needed on the data arc and z is the **buffer length** for the control arc.
2	LP.	Number of Logical Processors required for the problem.
3	Initializatio n Table.	[1 x LP x 2] Matrix to indicate initial node to Processing Element scheduling patterns. Each (x,y) entry indicates (node number, iteration-number).
4[a]	Scheduling Table : Next Node.	[1 x LP] table to indicate the next node scheduling pattern.
4[b]	Scheduling Table : Iteration Incr.	[1 x LP] table to indicate the iteration number increments.
5	Assignment Table.	[8 x LP] table to indicate node to physical element assignments for LP iterations.
6	Modulo Operators.	[1 x LP] table to indicate modulo (%) values for each node. AMOS uses these modulo values to compute a processor ID in the assignment table. They are the same as the number of processors tied to a circuit.
7	Maximum Iteration #.	The total number of iterations to be done. This is the number specified in the field plus one.
8	Node Delays.	The individual node execution times.

Table 3.1 Format of the Specifications File		
9	Drive Letters.	[LP * 3 characters] string table for drive letters for the RAM disks of Pes involved in the system.

Note : The expression [A x B] indicates a matrix with A rows and B columns.

Table 3.2 FDT File Format as Input to ATAMM Analysis Tool					
Time at which the FDT event occurred	Event Type	Node Number	Color (Simplex)	Processor Identification Number.	Iteration Number
[milli-seconds]	One of following: Reset Fire Node Control In Control Out Data in Process Begin Process End Data out Done node	[NODE#]	Always 1.	[PID_#]	[Iter. #]

Note : FDT Events are one of the following:

Reset : Indicates information about system reset.

Fire Node : Indicates time at which a node initiated execution.

Control Out : Indicates time at which control tokens were regenerated.

Control In : Indicates time at which control tokens were read in.

Data In : Indicates time at which data tokens were read in.

Process Begin : Indicates time at which execution of node code was begun.

Process End : Indicates time at which execution of node code was stopped.

Data Out : Indicates time at which data tokens were generated.

Done Node : Indicates time at which a node completed execution.

Halt : Indicates time at which a processor terminated its activities.

cyclo-static schedule and block cyclo-static schedule.

3.2.1 Cyclo-Static Schedule Example

The cyclo-static schedule for the example AMG follows the schedule shown in Figure 1.6. The specifications file for a cyclo-static schedule is shown in Table 3.3. The scheduling policy required three processors to execute the AMG. FDT data was recorded and the results of processing are shown graphically in Figure 3.1. These results have been extracted from the output display of the ATAMM Analysis Tool.

For this example, processor zero is scheduled to operate on the node sequence 0,1,3,2,4, beginning with node zero for iteration *i*. Processor two is scheduled to operate on the same sequence but it executes its initial node, node zero, for relative iteration *i+1*. Similarly processor one executes its thread of the sequence beginning with node zero for iteration *i+2*. This behavior of processors can be traced out in Figure 3.1 by following the hatched blocks that represent them. The sequence of node executions for the remaining processor may be similarly verified. The interesting aspect of cyclo-static operation as shown in Figure 3.1 is the mutual exclusivity and collective exhaustivity of the scheduling process that becomes apparent. Though each processor performs every node once in a 3 * TBO time frame, the relationship between node, processor and iteration is unique. Consequently, this ensures a deadlock free operation, as seen in Figure 4.1. A variation can be seen between the desired TBO of 5 and the actual TBO of 5.879 in Figure 3.1.

3.2.2 Block Cyclo-Static Schedule

Again, the reference AMG is that shown in Figure 1.1(b) with block cyclo-static schedule as expressed in Figure 1.7. The AMOS specifications for a block cyclo-static schedule appear in Table 3.4. Again, three processors were employed to execute the AMG and the execution results were recorded in a FDT file. Results of execution are shown in Figure 3.2. Note that processor zero executes nodes zero, one and four only in every 3 * TBO time frame. So does processor one, but for different relative iteration numbers. However,

Table 3.3 Specifications File for the Cyclo-Static Example

Field	Description
Connection Matrix	1,1,1 -,-,- -,-,- -,-,- 0,0,1 -,-,- -,-,- -,-,- -,-,- -,-,- -,-,- 0,0,1 0,0,1 -,-,- -,-,- -,-,- -,-,- -,-,- 1,1,1 -,-,-
Logical Processors	3
Initialization Table	0,0 2,0 0,1
Next Node	1 3 4 2 0 - - -
Next Increment	0 0 0 1 2 - - -
Assignment Table	0 2 1 0 2 1 1 0 2 0 2 1 1 0 2 - - - - - - - - -
Modulo Operators	3 3 3 3 3 - - -
Maximum Iteration	9
Node Delays	3.0 2.0 2.5 1.5 3.0 - - -
Drive Letters	E: F: G:

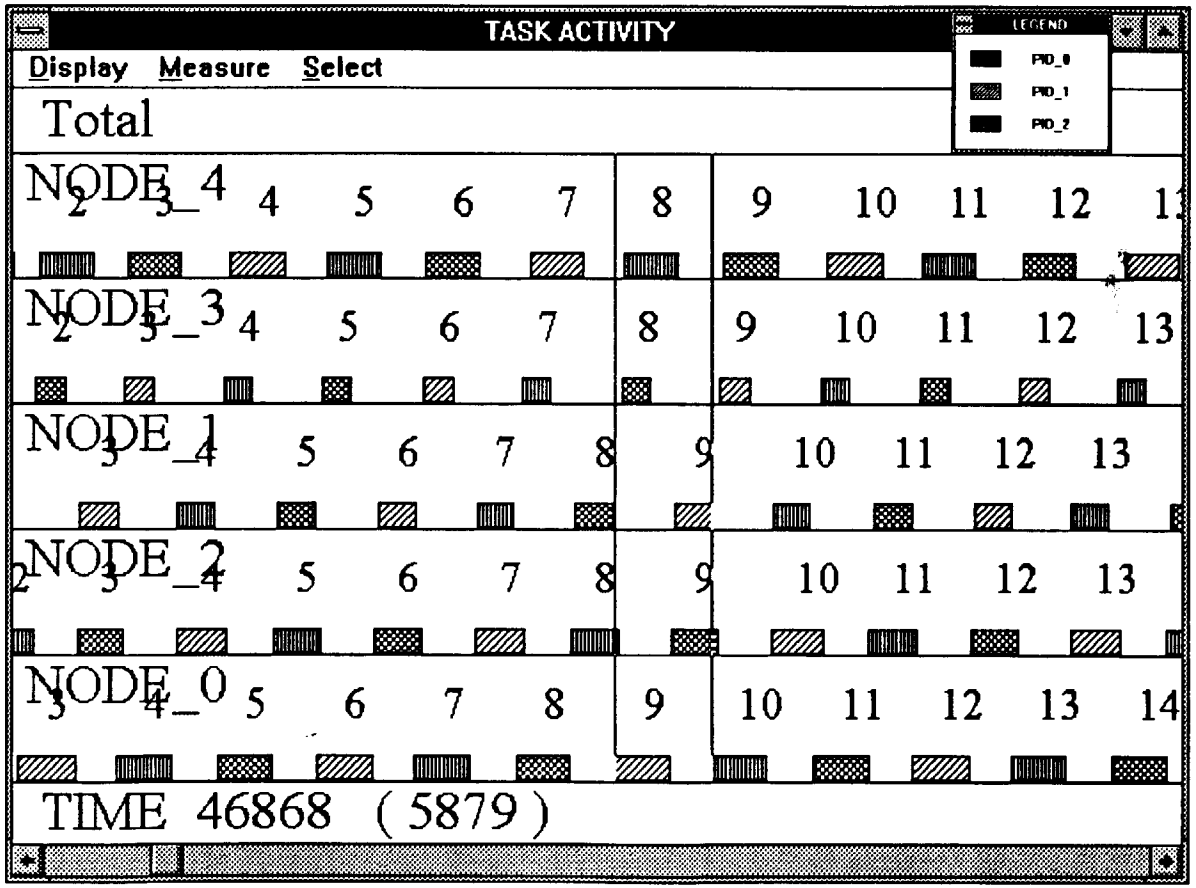


Figure 3.1 ATAMM Analysis Tool Output for Cyclo-Static Scheduling of the AMG in Figure 1.1(b).

Table 3.4 Specifications for Block Cyclo Static Example

Field	Description
Connection Matrix	-,-,- 0,0,1 0,0,1 -,-,- -,-,- -,-,- -,-,- -,-,- 1,1,0 -,-,- -,-,- -,-,- 0,0,1 -,-,- -,-,- -,-,- -,-,- -,-,- -,-,- 0,0,1 0,0,1 -,-,- -,-,- -,-,- -,-,- -,-,- 1,1,0 -,-,-
Logical Processors	3
Initialization Table	0,0 0,1 2,0
Next Node	1 4 3 2 0 - - -
Next Increment	0 0 0 1 2 - - -
Assignment Table	0 1 - 0 1 - 2 - - 2 - - 0 1 - - - - - - - - - -
Modulo Operators	2 2 1 1 2 - - -
Maximum Iteration	12
Node Delays	3.0 2.0 2.5 1.5 3.0 - - -
Drive Letters	E: F: G:

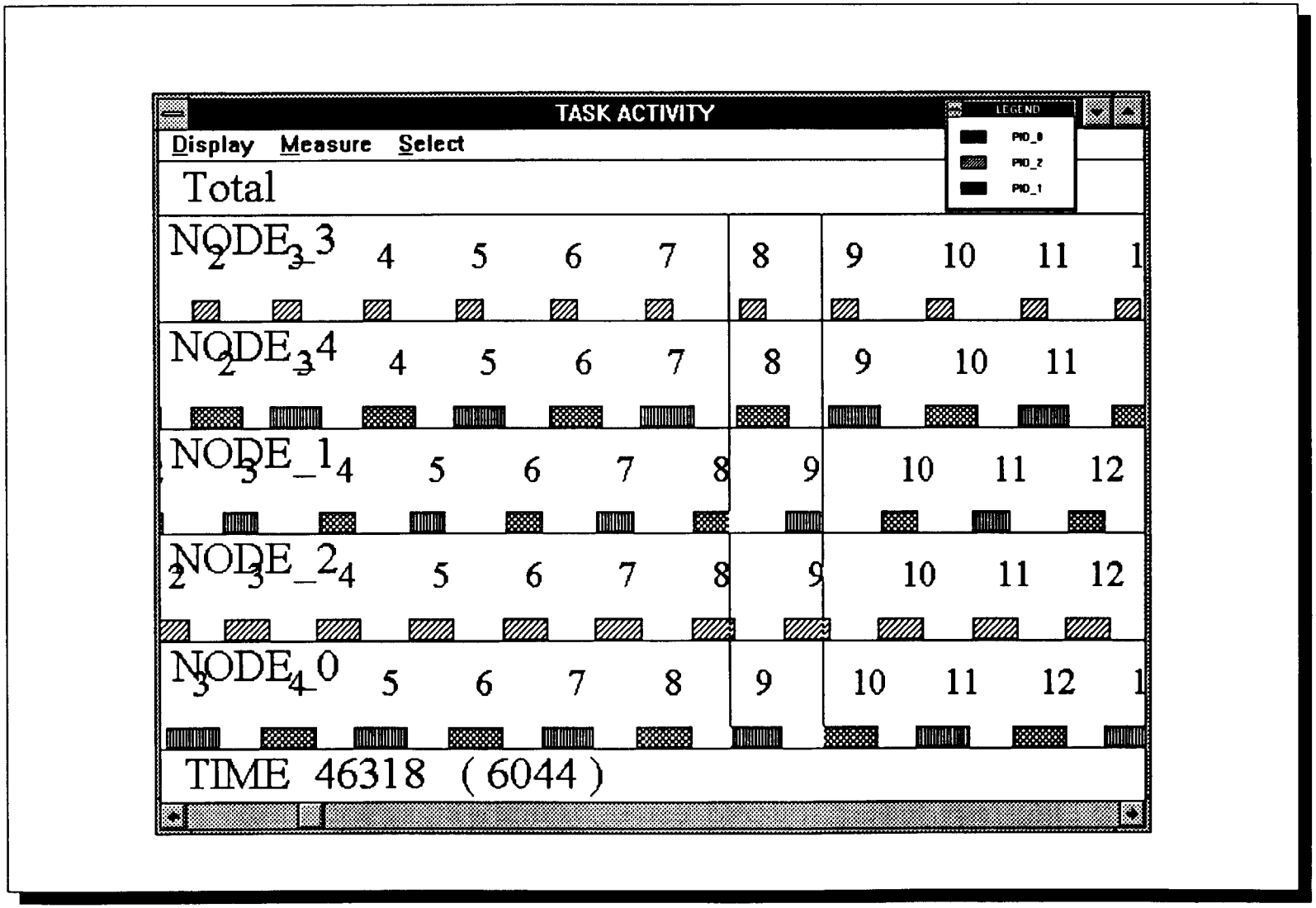


Figure 3.2 ATAMM Analysis Tool Output for Block Cyclo-Static Scheduling of the AMG in Figure 1.1(b).

processor two invariably switches its attention between executing nodes two and three.

The actual TBO for the block cyclo-static scheduling example is 6.044 instead of the ideal 5. However, despite this deviation from ideal behavior, the assignment and scheduling process is satisfied as seen from theexecution trace in Figure 3.2.

3.2.3 Static Scheduling Example

A possible static schedule for the example AMG has been shown in Figure 1.8. Specifications and results for a purely static schedule appear in Table 3.6 and Figure 3.3, respectively. Note that processor zero always performs nodes zero and one, processor one does nodes two and three while processor two is restricted to executes node four. The actual TBO is 6.153 now. As noted earlier this difference which can be attributed to the communication overhead shall be accounted for in Section 3.3.

3.3 Testbed Communication Overhead

The communication overhead of the testbed is governed by several factors. These include possible contention for network access or collisions due to simultaneous transmissions, measurement resolution limitations to the DOS timer's resolution of 55 ms and the observation that due to a single access ethernet channel, communication for a node grows linearly with the number of data interconnections that the node has with predecessor or successor nodes. The node execution may be expressed by

Total Node Execution Time =

Ideal Node Execution Time + Communication Overhead [3.1]

where

Communication Overhead =

Time to generate control tokens + Time to consume control tokens +

Time to consume data tokens + Time to generate data tokens. [3.2]

Table 3.5 Specifications for a Static Schedule Example

Field	Description							
Connection Matrix	-,-,-	0,0,1	0,0,1	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-
	1,1,0	-,-,-	-,-,-	-,-,-	0,0,1	-,-,-	-,-,-	-,-,-
	-,-,-	-,-,-	-,-,-	0,0,1	0,0,1	-,-,-	-,-,-	-,-,-
	-,-,-	-,-,-	1,1,0	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-
	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-
	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-
	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-
	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-
Logical Processors	3							
Initialization Table	0,0	2,0	4,0					
Next Node	1	0	3	2	4	-	-	-
Next Increment	0	1	0	1	1	-	-	-
Assignment Table	0	-	-					
	0	-	-					
	1	-	-					
	1	-	-					
	2	-	-					
	-	-	-					
	-	-	-					
	-	-	-					
Modulo Operators	1	1	1	-	-	-	-	-
Maximum Iteration	12							
Node Delays	3.0	2.0	2.5	1.5	3.0	-	-	-
Drive Letters	E:	F:	G:					

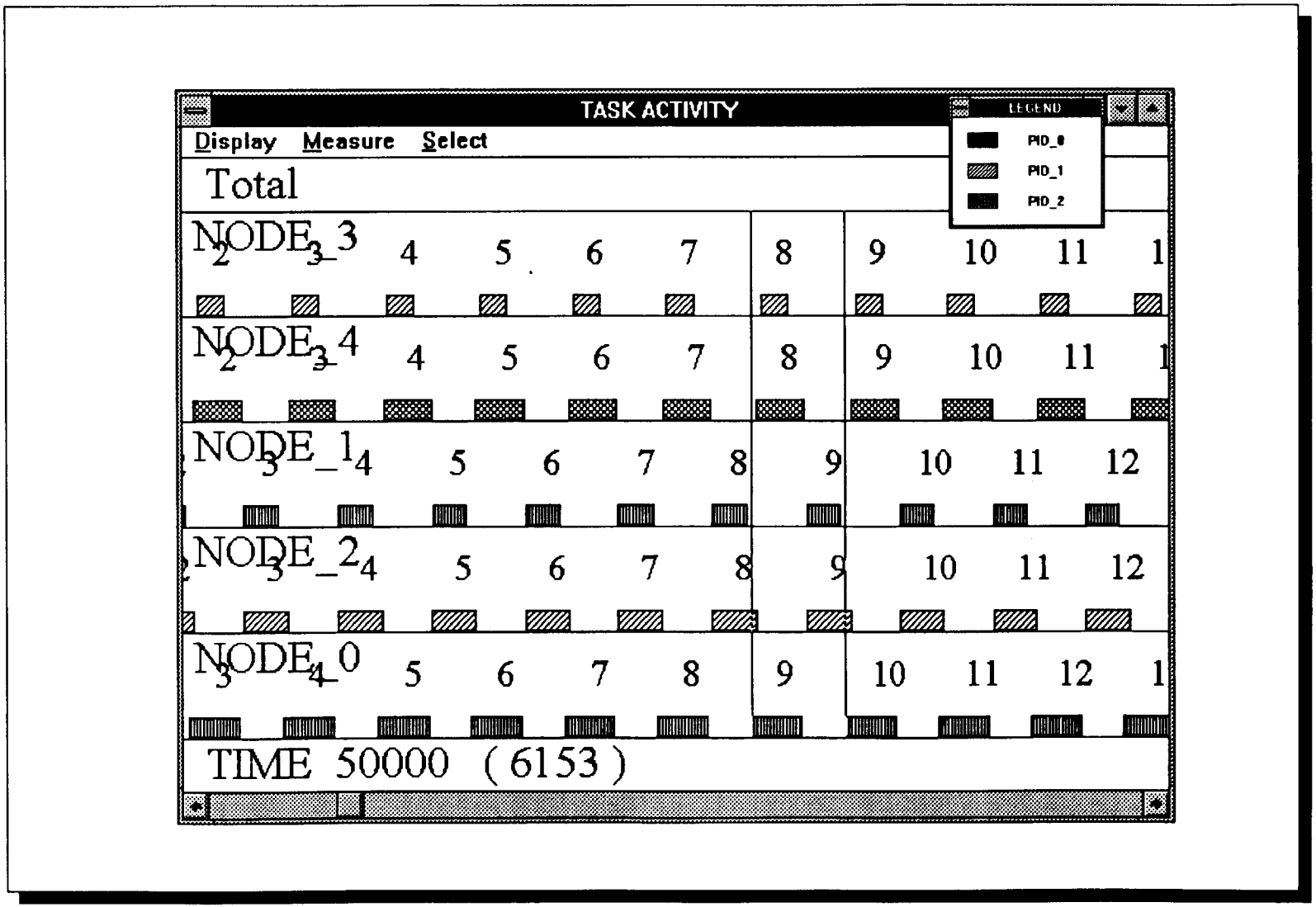


Figure 3.3 ATAMM Analysis Tool Output for Purely Static Scheduling of the AMG in Figure 1.1(b).

Minimum values for the communication components in equation 3.2 may be quantified further as,

Minimum time to generate control tokens =
Number of Control Tokens Output (NCO) * 55ms. [3.3]

Minimum time to consume control tokens = 1 * 55 ms. [3.4]

Minimum time to consume data tokens = 1 * 55 ms. [3.5]

Minimum time to generate data tokens =
Number of Data Tokens Output (NDO) * 55ms. [3.6]

It should be noted that the above time measures are absolute minimums. Due to contention or collisions, any of the above communication events may need additional 55ms network activity slots. Furthermore, the maximum size of an ethernet packet is about 1.5 KBytes. Hence tokens which exceed this size require two or more ethernet packet transmissions. There shall be a corresponding increase in the number of network activity slots required to complete the communication associated with the transmission of multiple ethernet packets. Moreover, the communication overhead increases linearly with the number of tokens that need to be generated per node.

In Figure 3.4, the TGP frame from Figure 3.1 has been magnified in order to display pertinent node execution and communication activities. Execution measurements for these figures have been tabulated in Table 3.6. These results account for the deviation from ideal behavior seen in the actual execution activity seen in Figures 3.6.

The critical circuit (which determines TBO) in the AMG in Figure 1.1(b), contains nodes zero and one. Consequently, TBO is determined by the execution of nodes zero and one. The ideal TBO for this AMG is 5.0 seconds (5000 milliseconds). Factoring in the communication overhead (determined through using Equations 3.1 through 3.6), a minimum TBO value of 5550 ms is expected. However, the actual TBO turns out to be 5879 ms. This

difference is believed to be due to contentions which occur due to simultaneous network access requests by peer processors. The figure of 5879 ms is derived by adding the actual cumulative execution times for nodes zero and one, which are 3517 ms and 2362 ms, respectively.

3.4 Summary

The distributed AMOS testbed successfully demonstrated various cyclo-static scheduling policies including cyclo-static, block cyclo static and static schedules. The demonstration AMG included such features as self loops, forwarded tokens, buffers on CMG control arcs. In additional exercises not reported herein, the testbed successfully executed AMGs requiring multiple instantiations of nodes. The execution of an eight node AMG on six processors demonstrated the upper operating limits of the testbed. Communication events occurring in the testbed are quantified using lower bound expressions that describe the minimum time a particular communication event may take. Though more effort is needed to refine the overhead computations, the testbed performed faithfully the underlying ATAMM model constraints. The distributed AMOS is significant in that graph managements was performed by only local knowledge of the graph requirements. It should also be noted that this was the first successful attempt to implement a purely distributed ATAMM based operating system.

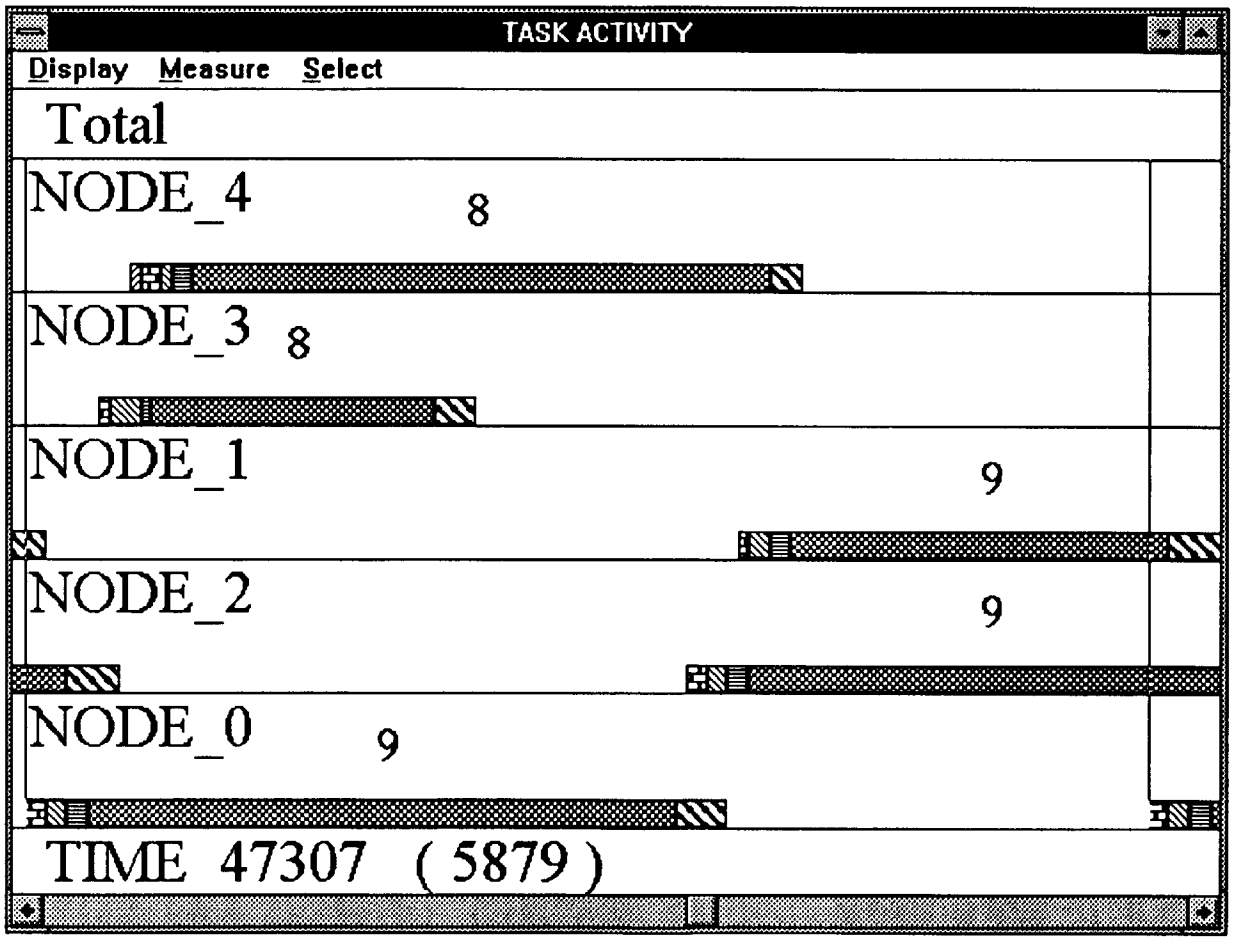


Figure 3.4 FDT Events in One TGP Frame of Figure 3.1.

Table 3.6 TBO Measurements for Figure 3.4.

CCr Node	Tokens & Timing (ms)	Output Ctrl Tokens	Input Ctrl Tokens	Input Data Tokens	Output Data Tokens	Node Time (ms)	Comm. Ov'hd. (ms)	Total Time (ms)	Idle Time• (ms)6	Min TBO (ms)	Actual TBO (ms)
0	Tokens	1	2	1	2					5550	5879
	Min.	55	55	55	110	3000	275	3275	0		
	Actual	55	110	110	165	3077	440	3517	0+#		
1	Tokens	1	2	1	2						
	Min.	55	55	55	110	2000	275	2275	0		
	Actual	55	110	110	165	1922	440	2362	0+#		

• Also includes the time interval between the termination of the execution of a node and the beginning of the execution of its succeeding node.
+ Experimental value approximated.
# Couldn't be determined accurately due to clock skew.
CCr Critical Circuit
Ctrl Control

CHAPTER FOUR
MICROCONTROLLER BASED AMOS

4.1 Introduction

An implementation an ATAMM dataflow multicomputer testbed using embedded firmware on microcontrollers employing a dynamic task scheduling strategy for distributed processing and presented in this chapter. Evaluation of the testbed is discussed in Chapter Five.

The natural progression in ATAMM research has resulted in an enhanced understanding of the ATAMM model and the simplicity of the control structure for AMOS. The regular structure of the ATAMM model and the control organization for AMOS has prompted this inquiry into the sufficiency of hardware based control for AMOS. Of interest is the development of a low level hardware based control structure for AMOS. Examining the efficacy and the constraints of this level of embodiment may provide insight for future implementation in imbedded wafer-scale-integrated (WSI) multiprocessors.

The testbed was designed using embedded firmware on Motorola 68HC11 microcontrollers. The testbed hardware consists of three microcontrollers (i.e processors) functioning as the processing elements and one processor performing the function of a centralized graph manager. Algorithm graphs are limited to eight nodes and, for purposes of control struction evaluation, graph nodes are timed only and do not perform any data operations. The AMOS control structure is based upon a message passing model. The bus organization is modeled after a token ring for contention free message passing. Evaluation of the testbed is performed by analysis of the event timing data generated by the execution of a data flow graph on the testbed.

4.2 Implementation of the ATAMM Model

The components necessary for implementation of the ATAMM model can be divided into physical and logical components. Physical components include:

1. a target architecture consisting of a number of processing elements;
2. a global memory which is distributed or shared among the processing elements; and
3. a communication network between the processing elements.

Logical components include:

1. a task scheduler to schedule node activities among the processing elements;
2. a communication layer between the processing elements; and
3. specifications for performance in terms of desired throughput, execution time, and the number of processing elements, R.

As previously stated, the ATAMM Multicomputer Operating System (AMOS) is a logical interface between the dataflow graph and the target architecture hardware. The components of AMOS include data structures containing the dataflow graph, specified operating parameters, and a graph manager.

The AMOS graph manager performs the scheduling operations of the operating system and may be classified as either centralized or distributed. A Centralized Algorithm Graph Manager (CAGM) performs graph management by maintaining a composite view of the CMG at every point of its execution. The Distributed Graph Manager (DAGM) concerns itself with the management of unique partitions of the CMG distributed among the processing elements. These partitions are unique but logically contiguous. A complete description of distributed graph management can be found in [ROY93] and was presented in chapters one through three of this report. An example of a centralized graph manager used in an implementation of the ATAMM model was imbodied in the

Westinghouse Electric Corporation Advanced Development Module (ADM). This type of graph manager was redundantly distributed among four processors in order to incorporate a degree of fault tolerance.

4.3 The Centralized Algorithm Graph Manager

The components of a CAGM are the AMOS data structures and the AMG (CMG). The graph manager uses information communicated to it by the PEs to update the CMG. For every node in the CMG, the CAGM checks the global memory for the presence of all required control and data tokens necessary to execute the node. Once the CAGM finds enabled nodes it assigns them, depending on priority if more than one node is enabled, to PEs from a processor queue of available PEs.

A state diagram description of a redundantly distributed centralized graph management employed in the Westinghouse ADM is shown in Figure 4.1. A functional unit (processor) starts in the idle state, and remains there until it finds its own identification label on top of the processor queue (first in first out). When it does, it enters the examine state and searches for enabled nodes in the CMG data structure. After finding an enabled node it removes itself from the top of the queue, updates the CMG, reads input data, broadcasts an "F" command to the rest of the PEs, and enters the execute state. The "F" command contains the updated CMG, updated processor queue, and the ID of the PE processing the CMG node. After the completion of node executiion, the PE writes the output data to global memory, updates the CMG, and broadcasts the "D" command which contains the updated CMG and the data generated by the execution of the node. Before returning to the idle state the PE may enter a test state where it performs a self test. This provides the means to remove a PE from the system during real time operation. If the PE is functioning correctly it will place its ID at the bottom of the processor queue and broadcast the "R" command with the updated processor queue to the other PEs. The broadcast of the "F", "D", and the "R" commands provides all the

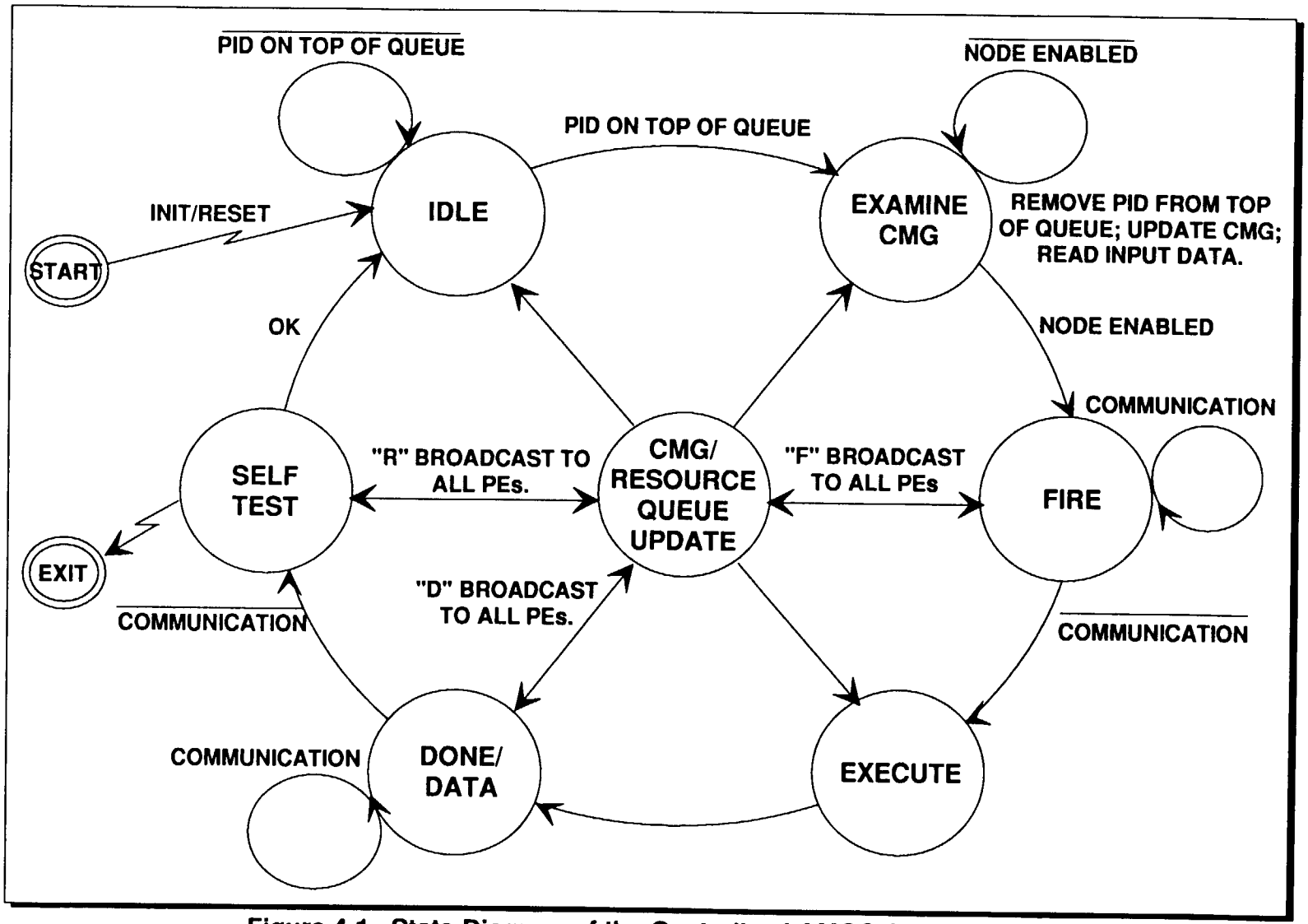


Figure 4.1. State Diagram of the Centralized AMOS Graph Manager.

status information necessary for the graph manager to maintain the status of the CMG.

Since the operation of the system is asynchronous, the graph manager must generally be interrupt driven in order to update graph status. When a broadcast is received a processing element, is interrupted from its current state, and enters the update state. It remains in this state long enough to update the CMG, global memory, and the processor queue as necessary.

4.4 Message Passing Model

Message oriented systems are characterized by facilities for passing messages among processes and for queuing messages at destination processes until they can be acted on [LAUER79]. The message passing style of system architecture is pertinent to real time systems and process control applications where the applications are encoded in message blocks.

An important consideration with any message passing system is in colision avoidence of the messages. AMOS can introduce contention for the communication channel (bus) when two or more processing elements complete a node operation at the same time. For instance, two processing elements broadcasting a "D" command need access to the bus. The implementation discussed in [ROY93] uses the IEEE 802.3 standard (CSMA/CD). In this scheme if two stations on the bus transmit data simultaneously, a collision is detected and the stations wait a random amount of time before transmitting again. In theory then, a station could be shut out from transmitting for an indefinite period of time. This type of non deterministic behavior is undesirable in real time systems. Of interest then, is a message passing model that is data driven and contention free to provide determinism is meeting required real time deadlines.

A useful message passing model to eliminate contention is the IEEE 802.4 Token Bus standard [STALLINGS91] and is used as the basis of the physical and logical layers in the present design. Physical and logical level

diagrams illustrating the model are shown in Figure 4.2. It is important to note that, though physically the topology of the network is a bus, it is logically a ring. That is, the stations are assigned positions in an ordered sequence, with the last member of the sequence followed by the first.

4.4.1 Token Ring Description

A control packet known as the token regulates each station's right of access. When a station receives the token, it is granted control of the medium for a specified time. The station may transmit one or more packets and may poll stations and receive responses. When the station is done, or time has expired, the token is passed on to the next station in logical sequence. The station possessing the token now has permission to transmit. Hence steady state operation consists of alternating data transfer and token transfer phases. A flow diagram representing this method is shown in Figure 4.3.

4.4.2 Token Ring Timing

The upper bound on the amount of time a station must wait before it can transmit can be determined from two parameters used to analyze token bus networks. Token holding time (THT) is the maximum time that a station can hold the token. Token rotation time is the maximum time that a token can take to circulate. Token rotation time can be calculated from THT at each station and the total number of stations N, according to the following equation,

$$TRT = THT * (N - 1). \quad (2.3)$$

The primary disadvantage of a token bus network lies in the complexity of the token bus algorithm when compared to other bus arbitration schemes like CSMA/CD. A second disadvantage is the overhead involved under conditions of a light load in which a station may have to wait through many fruitless token passes for a turn to transmit.

One of the advantages of token bus is that its behavior is deterministic and contention free. The upper bound on the amount of time a station must wait before it can transmit is known because each station can only hold the

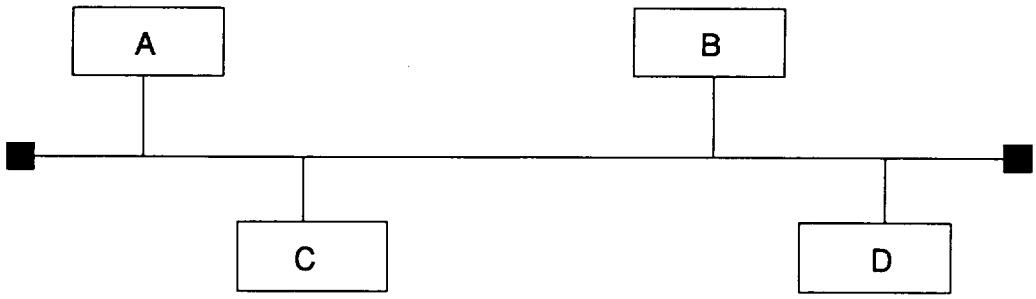


Figure 4.2(a). Physical Topology of Token Bus.

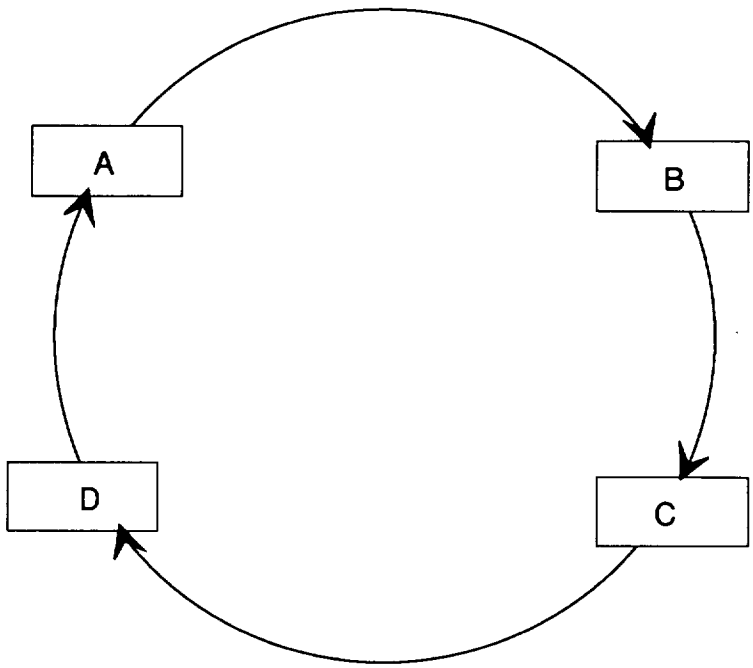


Figure 4.2(b). Logical Layout of Token Bus.

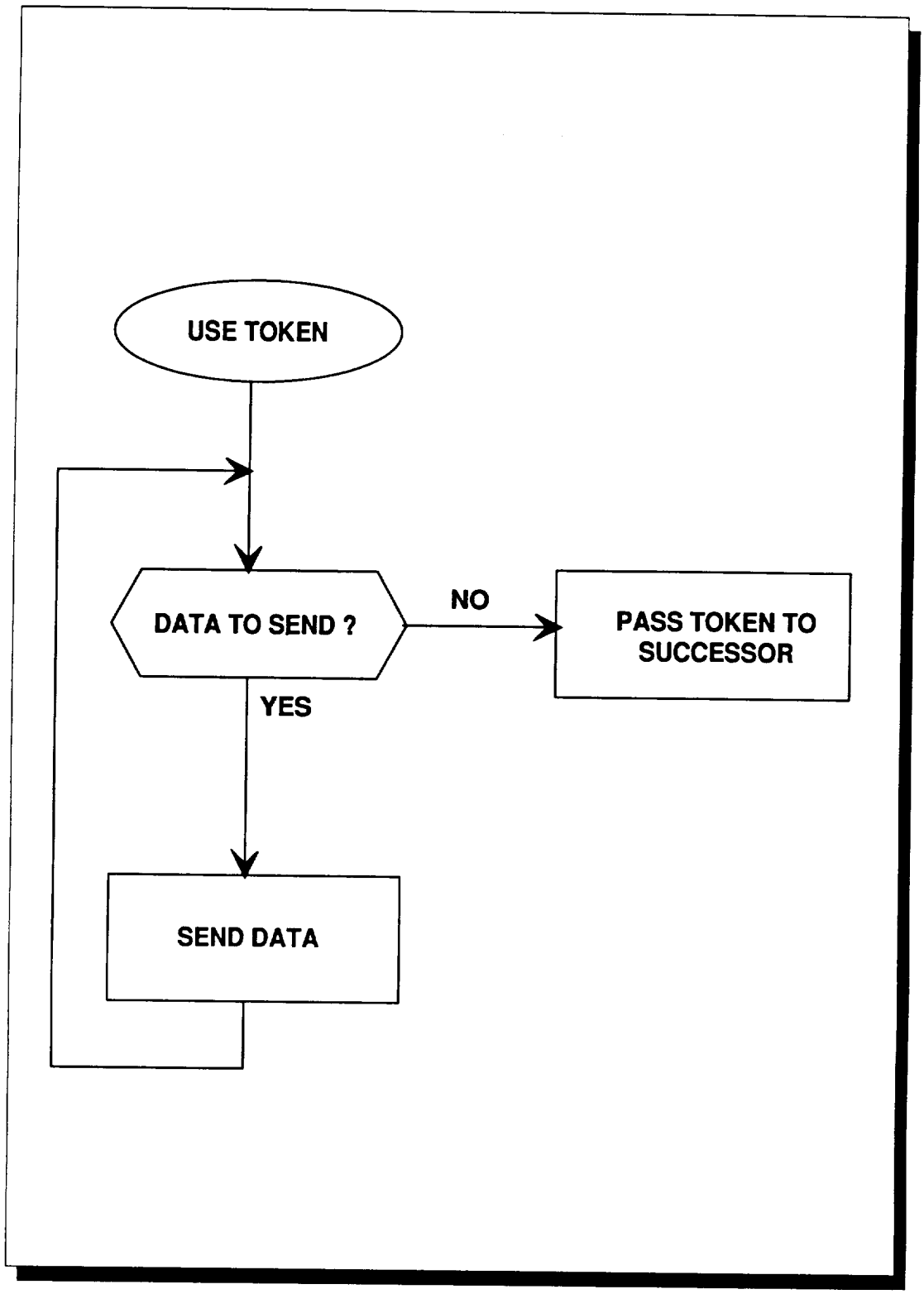


Figure 4.3. Flow Diagram for Token Bus Operation.

token for a specified amount of time. Thus, the determinism supports the token bus as a viable and attractive alternative to the CSMA/CD bus for real time applications.

4.5 Task Scheduling for AMOS

The main objective in the design of the testbed is to develop contention free task scheduling for AMOS at the hardware level. The Motorola 68HC11 microcontrollers was chosen for the implementation because of availability and their representation of a class of microcomputers that allow access to the CPU level hardware in a transparent manner to the user. Some of the features of multicomputer operating systems [HWANG84] that are relevant to AMOS include:

1. an initialization and synchronization mechanism that ensures orderly execution of all system operations;
2. a communication manager to handle interprocessor communication between multicomputer PEs;
3. a task scheduler that schedules tasks that are ready for execution with available processors in a manner that prevents deadlocks and avoids abnormal program termination; and
4. a resource manager that allocates, removes and manages computing resources(PEs) within the system.

These characteristics are readily visible in prior imbodiments of AMOS which have been message passing in nature. Consistent with the design of good message passing operating systems are the following characteristics [GORSLINE86]:

1. synchronization among processes and queuing for unavailable resources is implemented in the message queues attached to the processes associated with those resources;
2. data structures that must be manipulated by more than one process are passed by reference in messages;

3. no process touches the data unless it is currently processing a message referring to them, and a process does not continue to manipulate data after it has passed the data to another process via a message;
4. peripheral devices are treated as processes (or virtual processes) for which control often resembles sending a message to that device, and an interrupt from that device is manifested as a message to some other process; and
5. processes operate on a very small number of messages at a time and normally complete the operations necessitated by these messages before looking at the message queues again.

4.6 Communication System Design and Implementation

A description of the interaction between the logical and physical components used for implementing inter-PE communication is presented in this section.

4.6.1 Communication Layer Overview

A representation of a system with four functional units is shown in Figures 4.4, 4.5, and 4.6, respectively. In Figure 4.4, one of the functional units is the graph manager and the others are assigned to node operations. Each functional unit has a logical communication layer and two message queues. One of the message queues is for outgoing messages and the other is for incoming messages. The communication layer is a resident logical layer in each functional unit that allows the functional units to communicate with one another. The figure also shows the "hard" message passing and the "soft" message passing aspects of the message passing model. Hard messages are those messages being passed from one functional unit to another using hardware. Soft message are those messages exchanged between software processes (logical layers) in a functional unit.

A physical interpretation of the logical layer shown in Figure 4.4 is

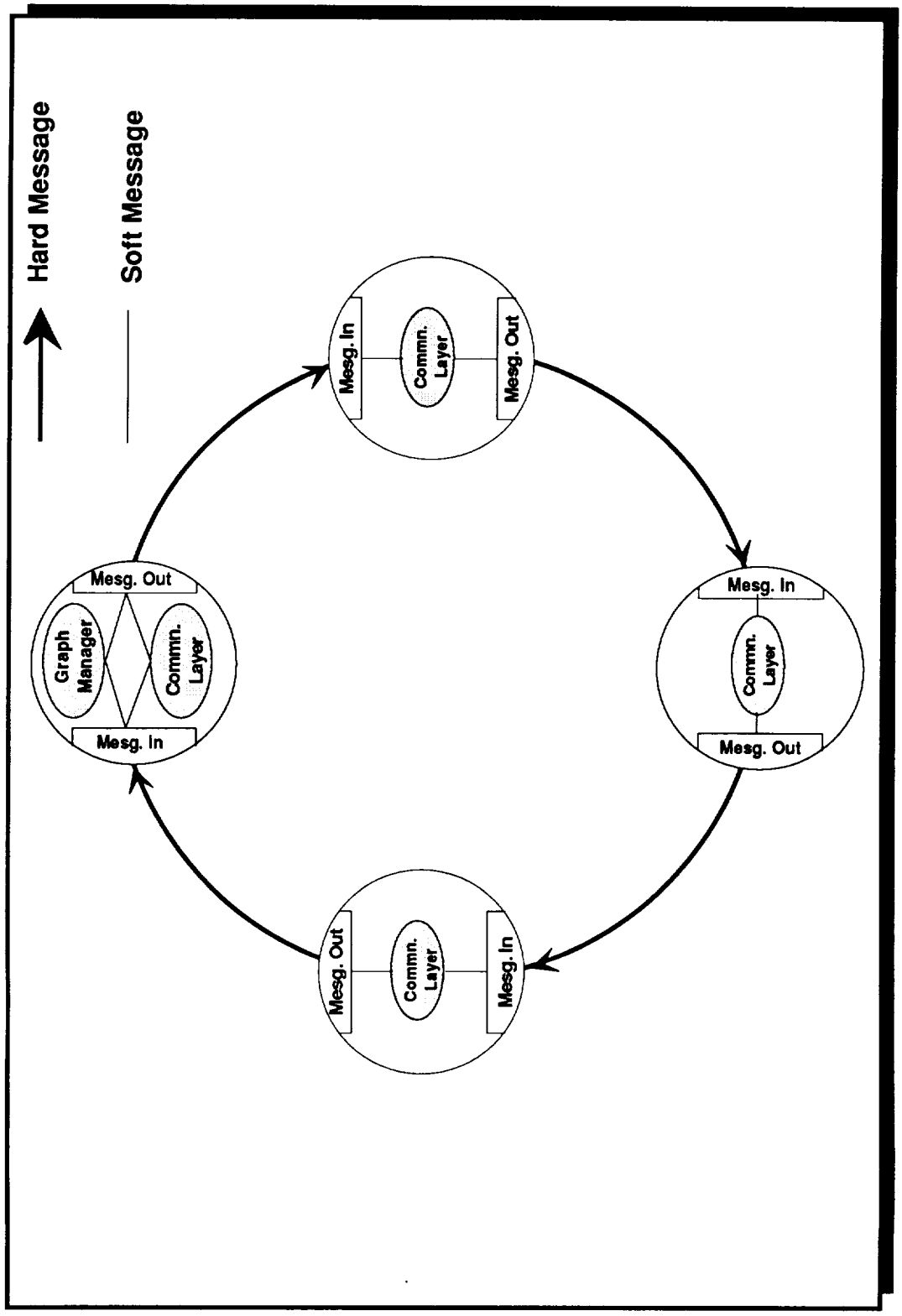


Figure 4.4. Logical Representation of the Message Passing Model.

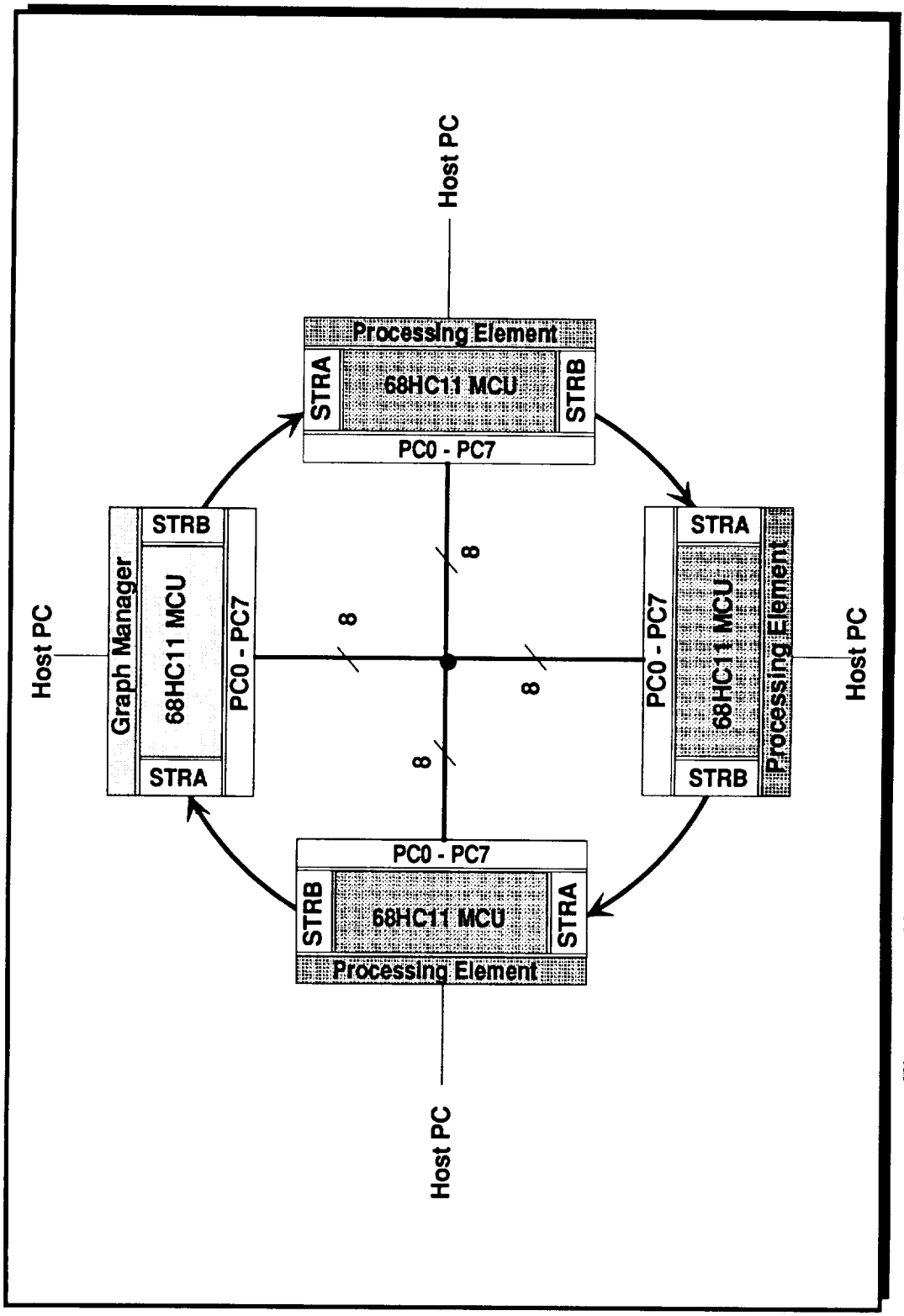


Figure 4.5. Hardware Representation of the Message Passing Model.

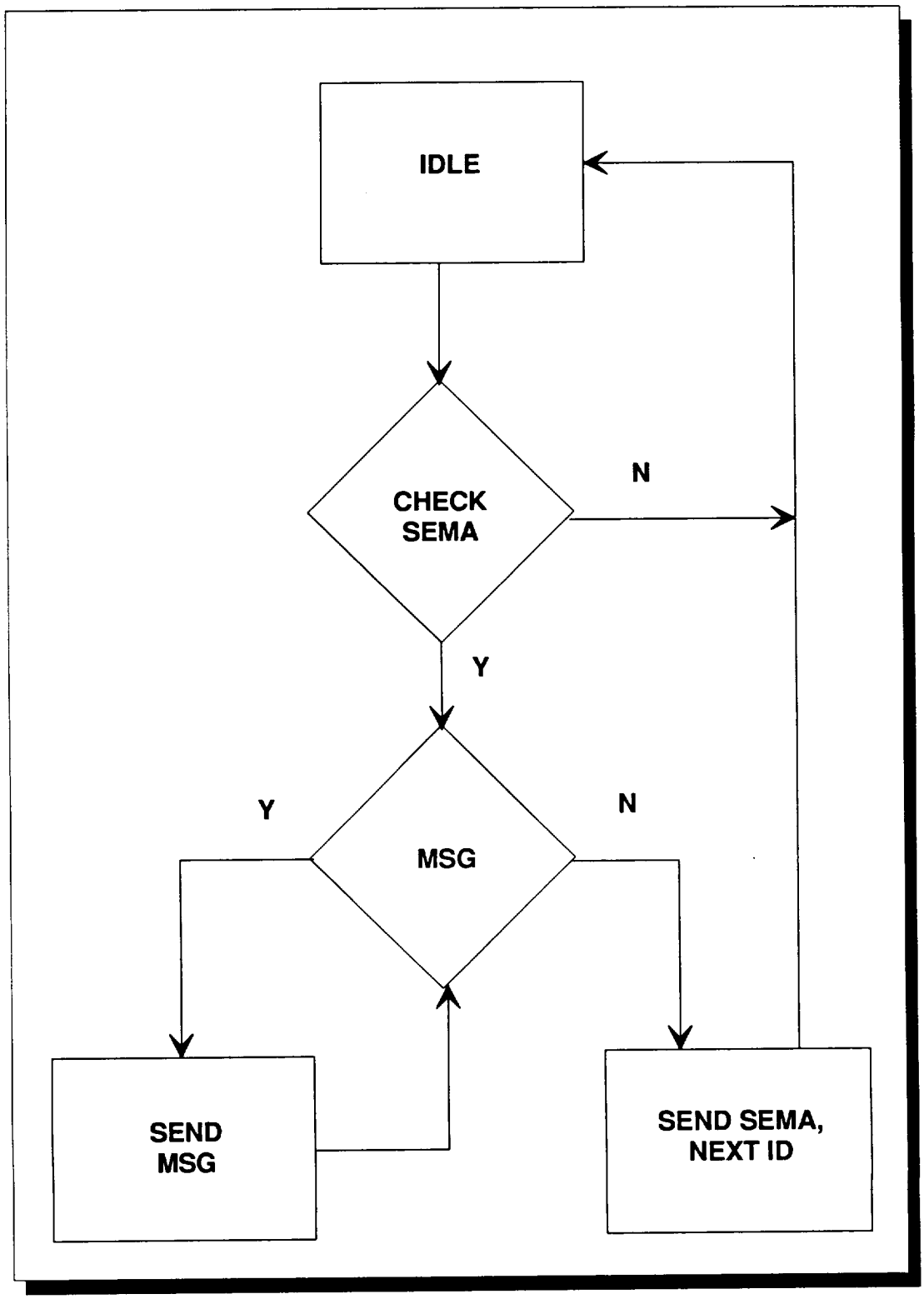


Figure 4.6. Flowchart Representation of Token Bus Operation.

shown in Figure 4.5. PEs communicate with each other over a bus. The analogy between the logical and physical layer is clearly seen by comparing Figures 4.4 and 4.5. Soft message passing occurs between software processes in the software resident in each of the PEs. Hard message passing occurs when one PE sends a message to another PE. The Host PC shown in Figure 4.5 acts as an interface to the microcontroller.

4.6.2 Communication Layer Design

The message passing model requirements are to pass messages easily and effectively between the functional units using a token bus. Since AMOS operation is asynchronous, the message passing is interrupt driven. The medium access technique chosen as a model was the IEEE Token bus standard (802.4). While this particular implementation does not conform to the Token bus standard in its entirety, it does follow it closely enough to use the IEEE 802.4 standard as the model.

The logical configuration of the token bus is a ring. Each PE is assigned a unique identification number(ID). During ring initialization, one of the PEs is granted access to the bus (ie, holds a semaphore) and can send a message to the other PEs. The PE relinquishes control of the bus by passing the control packet known as a semaphore with the ID of the next PE in logical sequence. A flow diagram representing this sequence is shown in Figure 4.6. As shown in that figure, each PE is in one of two states:

1. waiting for the semaphore, or
2. transmitting a message or sending a semaphore with the ID of next PE.

Communication takes place in the form of alternating data and semaphore transfers. When a PE has the semaphore it may transmit data; else it waits for the semaphore. This constitutes the logical layer of the token bus.

4.6.3 Physical Layer Design

Translation of the logical layer into hardware was influenced by the constraints imposed by the selection of the 68HC11 microcontrollers which is

an eight-bit architecture. One of the major constraints was that there was only one port available (port C) that provided full bidirectioanal asynchronous handshaking for parallel I/O. An eight bit wide bus is formed by connecting all port C lines from the four PEs together. This forms the network bus on which passes all the message transfers from one PE to another. Each message transfer is interrupt driven and takes place asynchronously as is illustrated in Figure 4.7.

The use of port C also entails the use of handshaking lines STRA and STRB and provide acknowledgement when a PE sends a message. The STRA and STRB lines are shown in Figure 4.7. When PE A writes to port C, and hence to the bus, it causes its STRB line to go low. The STRB line of PE A is connected to the STRA line of the next PE (B) in logical sequence and causes an interrupt in PE B. When B reads the bus in its interrupt service routine (ISR), its STRB line is driven low, and causes an interrupt in the next PE in logical sequence since B has its STRB line connected to STRA of the next PE, and so forth. This continues until A receives an interrupt from D which serves as the acknowledge for the message that A put on the bus and A can now write to the bus again if it has a message to transmit. If not, A passes on the semaphore with the ID of the next PE in logical sequence. Thus all messages are acknowledged by the hardware hanskaking process.

4.7 Design of the Graph Manager

The graph manager was designed with consideration of the data structures required to implement the state diagram of the graph manager and attendent message passing.

4.7.1 Overview of Graph Manager and PE interaction

Graph management in this implementation of the ATAMM model is centralized but not redundantly distributed as in the ADM version described in Chapter Two. The most important conclusion to be drawn from this view

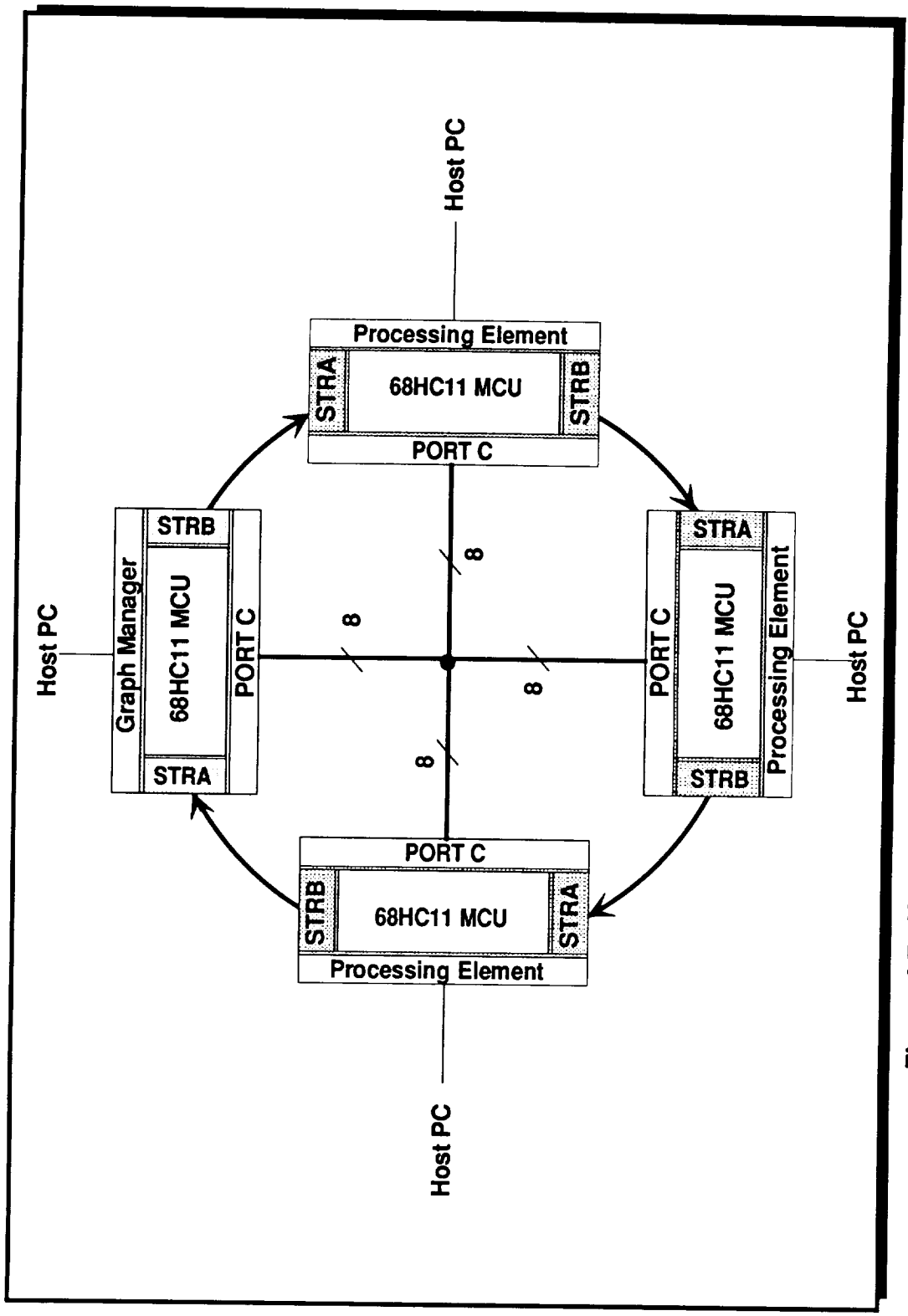


Figure 4.7. Handshaking Scheme for the Message Passing Model.

of graph management is that only one PE functions as the graph manager. The consequences of this approach are that the PEs have to transmit and respond only to a limited number of messages. The graph manager has to transmit "Fire" commands and respond to "Done" commands. The other PEs have to transmit "Done" commands and respond to "Fire" commands. To carry out token bus management every PE has to respond to semaphores addressed to it and transmit semaphores with the ID of the next PE in logical sequence. State diagrams illustrating this behavior are shown in Figures 4.8(a) and 4.8(b). It can be seen that the state diagram of the centralized graph manager described in chapter two (Figure 2.8) can be reduced to a two state diagram for the graph manager and a two state diagram for each of the other PEs. This reduction in states is possible since graph management duties are not redundantly distributed and the graph manager does not take part in node operations.

4.7.2 Message Format

The discussion of the message passing model to this point has not addressed the format of the messages to be passed. The Motorola 68HC11 is an eight bit microcontroller and this imposes a physical constraint on the number and the format of the messages that can be passed. Messages to be passed are of three types, Fire, Done, and Semaphore. Fire and Done messages have to include a processor ID (PID) and a node number while the semaphore only has a processor ID. The command word (CWD) is restricted to eight bits since the bus is eight bits wide. The three commands required, Fire, Done, and Semaphore require two bits for encoding in the CWD. Assuming a maximum of eight nodes for the purposes of this research, three bits are required for encoding the node number. The three remaining bits are used to encode the processor ID. This gives the testbed a capability of a maximum of eight nodes for the graph and eight processors to execute it.

4.7.3 Enhanced State Machine view of CGM

The state diagrams in Figure 4.8 provide a top level view of the

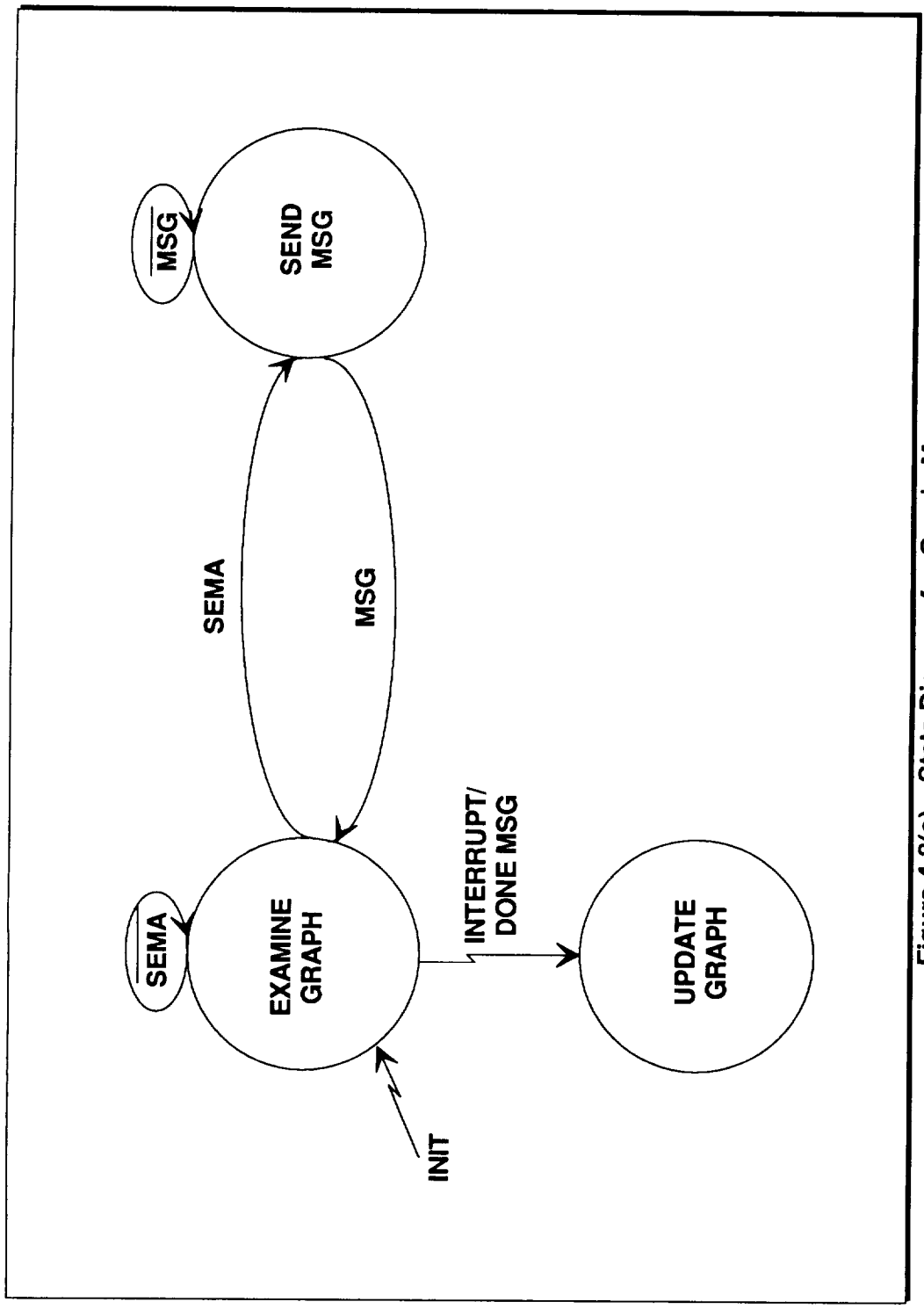


Figure 4.8(a). State Diagram for Graph Manager.

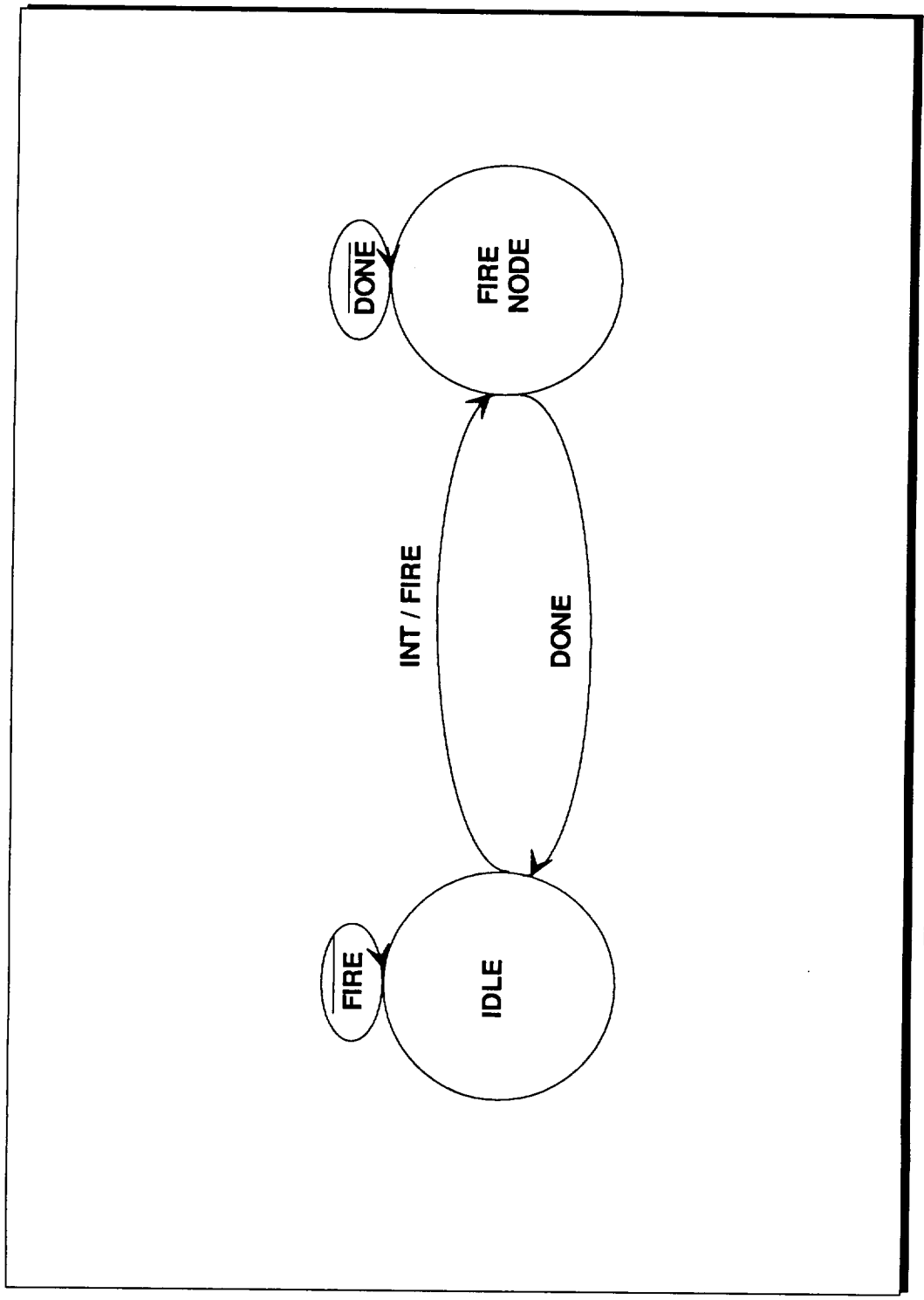


Figure 4.8(b) State Diagram for Processing Elements.

functional aspects of the graph manager and the other PEs, but do not provide all the information about the data structures required to implement the graph manager. An enhanced state machine diagram of the graph manager is shown in Figure 4.9.

The state diagram in Figure 4.9 describes the message passing activities of the graph manager and associated data structures. On inspection of Figure 4.9, the following logical structures can be identified:

1. a static data structure containing information pertaining to data and control edges;
2. a dynamic data structure to hold information about the current state of execution of the graph ie, current position of data and control tokens;
3. an outgoing and incoming message queue to hold messages from and to the other PEs; and
4. a processor queue to hold information about the availability of PEs for node operations.

4.7.4 Software Design

The philosophy behind the implementation of the state diagram for the graph manager in Figure 4.9 is to develop a message passing software state machine. The program consists of a main routine and four subroutines. The main routine consists essentially of the communication layer discussed in section 3.2. The graph manager consists of four subroutines that perform the following functions:

1. Examine Graph. This subroutine examines the graph for enabled nodes and passes a message with the enabled node number to the Examine Processor queue subroutine.

2. Examine Processor Queue. This routine inspects the processor queue and appends the ID of an available PE to the message passed

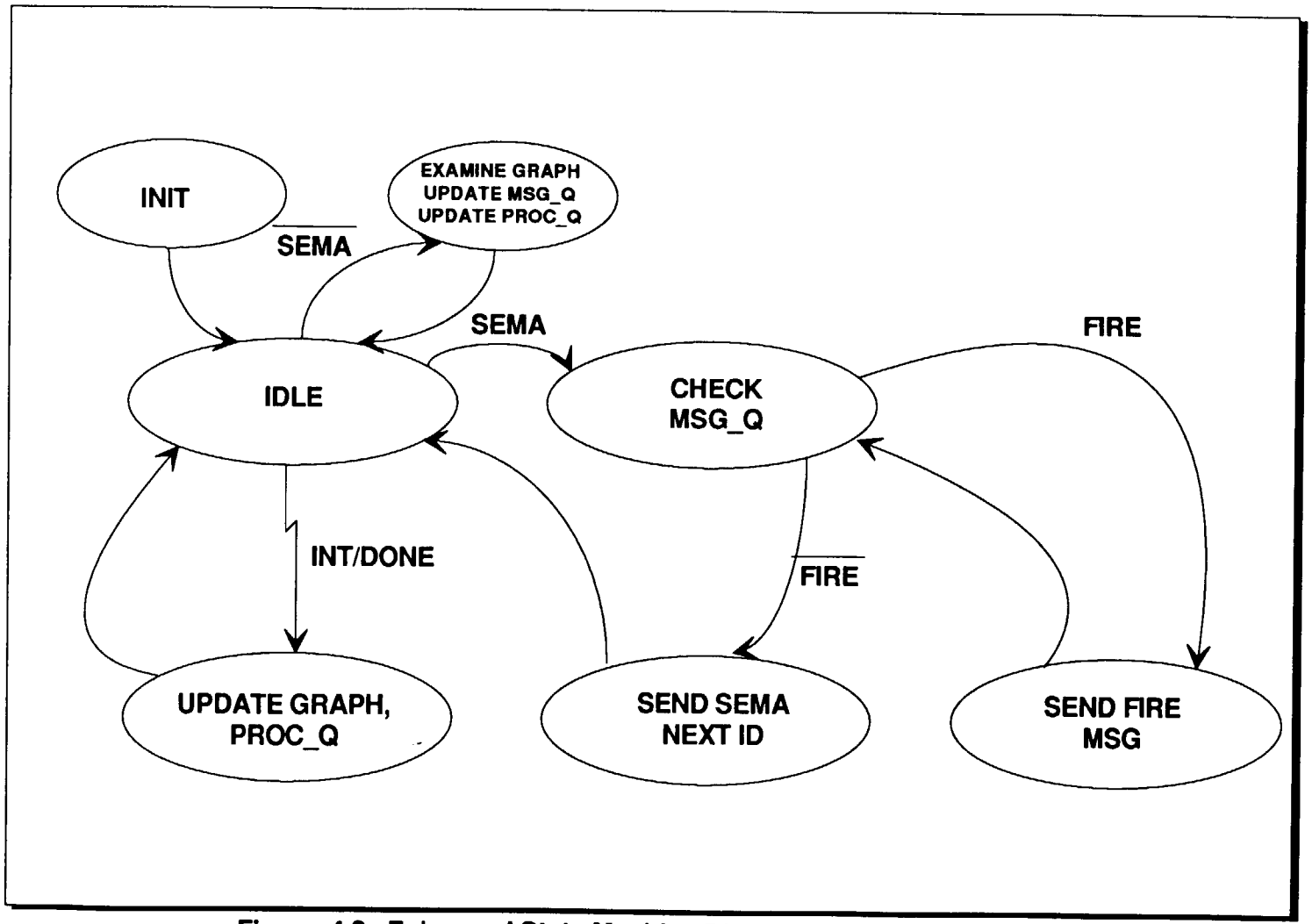


Figure 4.9. Enhanced State Machine Diagram for Graph Manager.

to it by the Examine Graph subroutine and puts a message into the outgoing message queue.

3. Update Graph. This routine has a message passed to it by the ISR and it extracts the node number of the node that is "Done" and updates the graph. It also calls and passes a message to the Update Processor Queue subroutine.

4. Update Processor Queue. This subroutine extracts the ID of the PE that sent a "Done" message from the message passed to it by the Update Graph subroutine and updates the processor queue.

The interaction between these subroutines and the main communication layer is illustrated in Figure 4.10. The message passing structure of the graph manager facilitates a degree of modularity than is naturally incorporated in its implementation.

4.8 Software Implementation

The software implementation of AMOS is presented in this section. Implementation of the graph manager is examined initially followed by a description of the resident software in each of the other PEs and the simulation of node activities using the 68HC11's internal clock.

4.8.1 Graph Manager Implementation

Software implementation of the graph manager consists of a translation of the data structures mentioned in the previous section into a form consistent with the eight bit target architecture and undderlying instruction set.

The data structures identified in section 4.7.3 can be implemented by the use of bit mapped arrays. The CMG information is representedin a set of four connection matrices in the form of eight by eight arrays (eight bytes by eight bits) which specify the static data edges of the CMG, the implicit control edges, current state of data tokens on the CMG, and the current state of control tokens in the CMG. An AMG example is shown in Figure 4.11. Related matrices for the example is shown in Tables 4.1 through 4.5.

C-2.

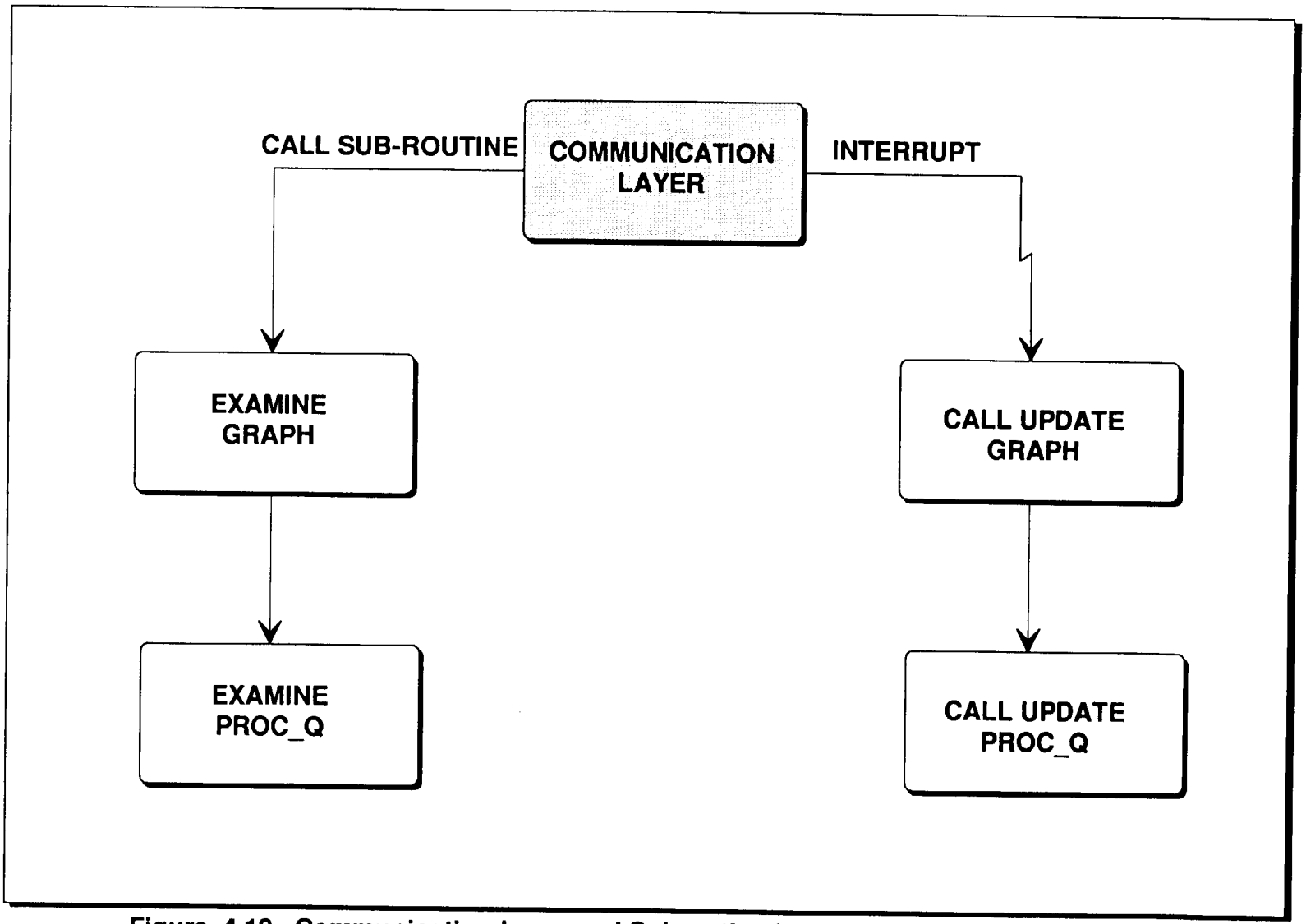


Figure 4.10. Communication Layer and Subroutine Interaction for the Graph Manager..

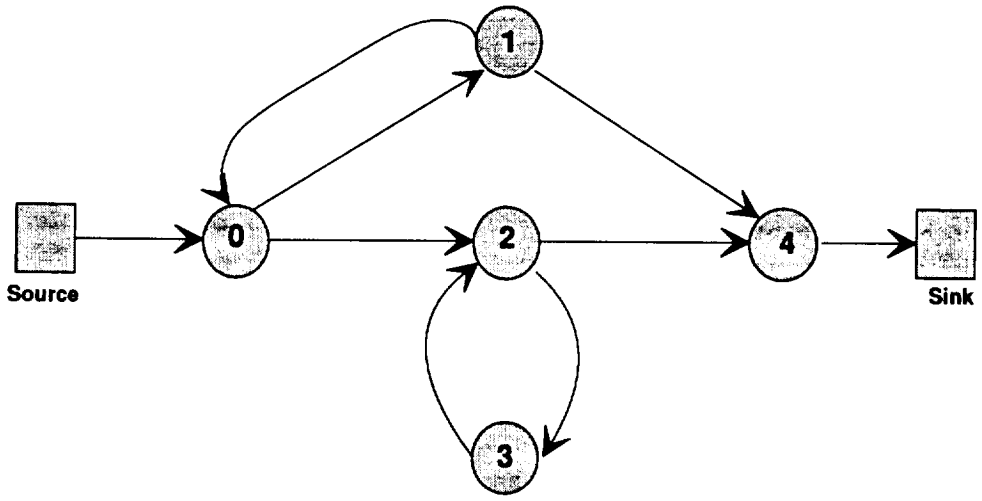


Figure 4.11(a). An Example Algorithm Directed Graph (ADG).

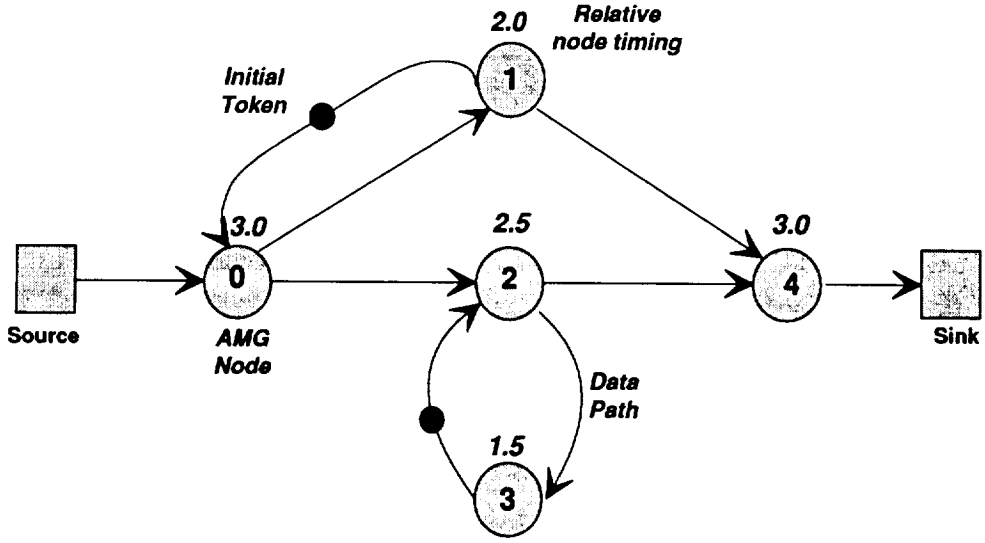


Figure 4.11(b). An Example Algorithm Marked Graph (AMG).

Table 4.1 Static AMG Mask

	0	1	2	3	4	5	6	7
0		1	1					
1	1				1			
2				1	1			
3			1					
4								
5								
6								
7								

Table 4.2 Transpose of Static AMG Mask.

	0	1	2	3	4	5	6	7
0		1						
1	1							
2	1			1				
3			1					
4		1	1					
5								

Table 4.2 Transpose of Static AMG Mask.

6								
7								

[] indicates the absence of an arc; 1 indicates an arc from row to column nodes.

Table 4.3 Initial Data Tokens.

	0	1	2	3	4	5	6	7
0		1						
1								
2				1				
3								
4								
5								
6								
7								

Table 4.4 Initial Control Tokens

	0	1	2	3	4	5	6	7
0		1	1					
1	1				1			
2				1	1			
3			1					
4								
5								
6								
7								

Table 4.5 Processor Queue Representation							
Pointer Mask							
0	0	0	1	0	0	0	0
Processor Mask One							
1	0	0	0	0	0	0	0
Processor Mask Two							
0	1	0	0	0	0	0	0
Processor Mask Three							
0	0	1	0	0	0	0	0

Tables 3.1,3.2,3.3 3.4 (4.1,4,2,4.3,4.4)

The first one in Table 4.1 represents the data edges for the AMG of Figure 4.11. It is constructed by row indices representing the predecessor nodes and column indices representing successor nodes. A "1" in a cell of the matrix indicates a data arc from row index (predecessor node) tocolumn index (successor node) in the AMG. The transpose of this matrix in Table 4.2 represents the implicit control arcs that exist from each successor node to each predecessor node. The current state of data and control tokens during execution are shown in Tables 4.3 and 4.4.

A set of four single row arrays to represent the processor queue. Each array indicates the current position of each processor in the processor queue, with one array that is used as a pointer. The implementation of the processor queue is shown in Table 4.5. The position of the "1" bit in the arrays indicates the position of the PE in the processor queue. The PE on top of the queue has a "1" in the most significant bit of the array corresponding to it. In Table 4.5, processor one is on top of the queue as shown in Processor mask one. Processor two is the next in the queue and Processor three is the last in the queue.

A third array provides a message table. The exact manner in which these arrays are used is discussed in the next section when an example CMG is modeled for execution.

A flow chart representing the operation of the graph manager and its relationship with the software processes in the other PEs is shown in Figure 4.12. The flow chart clearly shows the correlation of the software with the state diagram in Figure 4.9.

4.8.2 Implementation of the state diagram for node operations

The state diagram of Figure 4.8(b) describes the activities of the processing elements other than the graph manager. Since the graph manager does not perform any node operations it does not include the software layer required to perform node operations.

The PEs other than the graph manager perform all the node operations

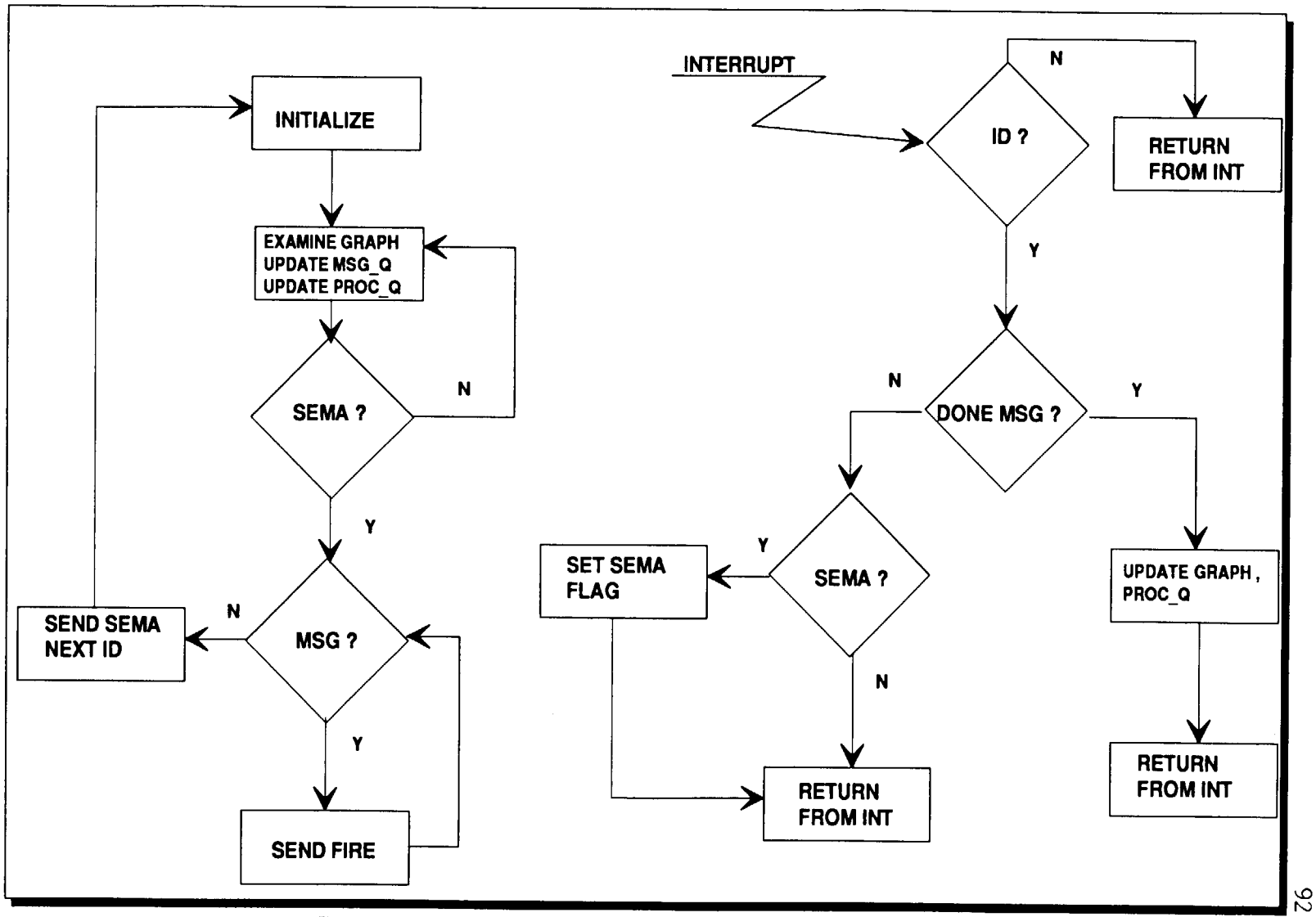


Figure 4.12. Flowchart for Graph Manager Operation.

required to execute the CMG. The operation of the PEs assigned to node operations follow the state diagram of Figure 4.8(b). A flow diagram for this action is shown in Fig 4.13. Initially all PEs are in an idle state. When a PE receives a "Fire" command and its ID the node number is decoded from the message and it proceeds with the execution of that node which is done asynchronously through an interrupt. When the PE receives a semaphore and its ID, the outgoing message queue is checked for a message (Done) and puts the message on the bus. If the message queue is empty it passes on the semaphore with the ID of the next PE in logical sequence.

4.8.3 Simulation of node timings

The node timings are simulated by creating a data structure that represents the node times. This data structure is distributed across all the PEs since the scheduling of node operations is dynamic and each PE can be scheduled for any node operation.

The 68HC11 has a real time interrupt (RTI) function that can be used to time the interval between two events. When the RTI is enabled, it requests interrupts at a rate of 4.1, 8.19, 16.38, or 32.77 milliseconds depending on the rate chosen. Thus, node timings corresponding to a multiple of one of these values for node operations can be simulated. When a PE receives a "Fire" message it decodes the node number from the message and uses it to point to an integer value in a look up table which corresponds to the data structure previously mentioned.

4.8.4 Memory Utilization

An important observation at this point is that the entire control organization for AMOS was implemented with the memory available on the 68HC11 microcontroller and without additional memory. The net amount of program memory available on each microcontroller was 768 bytes, of which 742 bytes were used for the 68HC11 that contained the software for the graph manager and 560 bytes were used for each of the PEs.

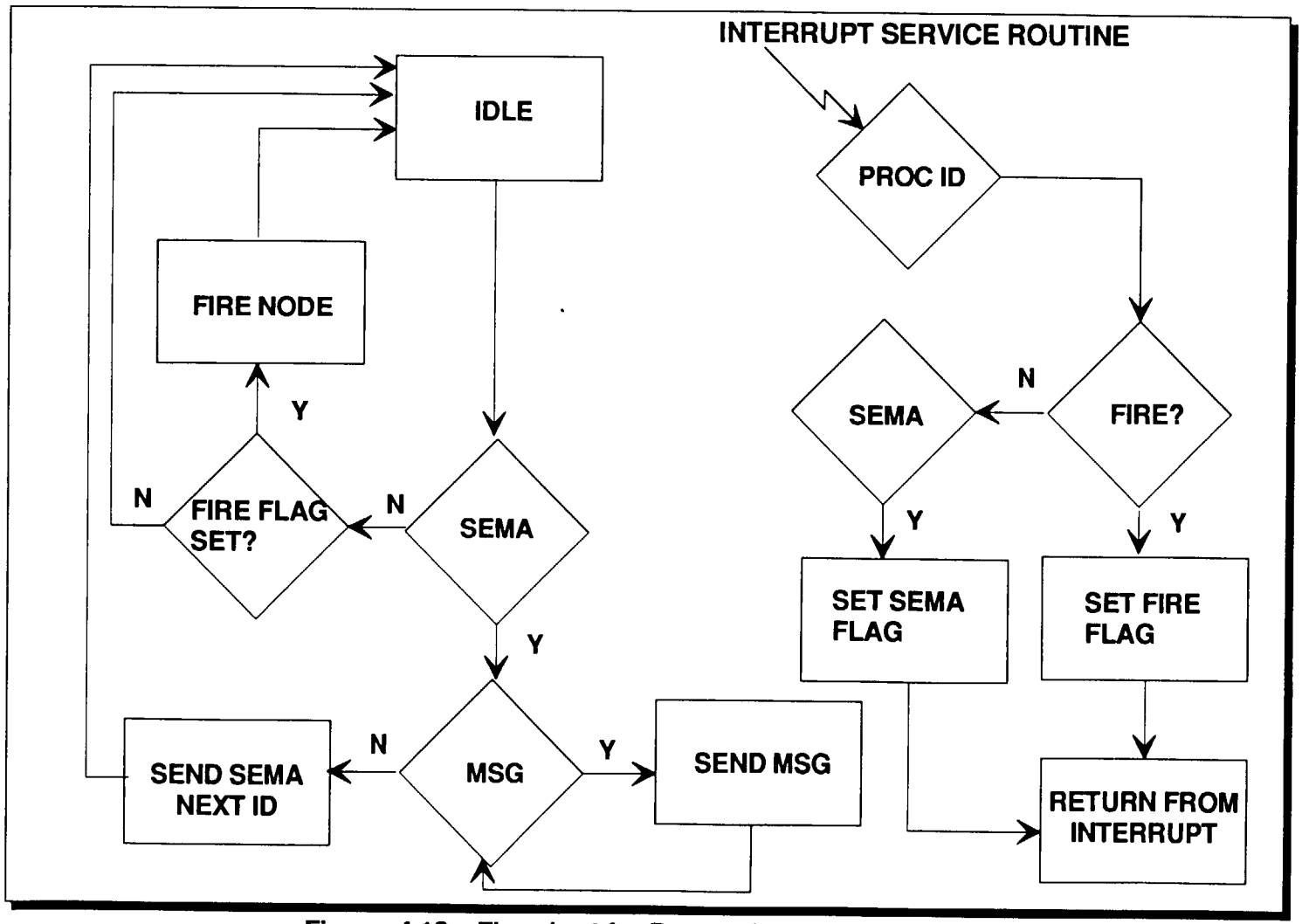


Figure 4.13. Flowchart for Processing Element Operation.

CHAPTER FIVE
TESTBED INTEGRATION AND EVALUATION

5.1 Testbed Operation

The example AMG of Figure 4.11 is modeled and executed to demonstrate the operation of the system. The AMG is represented in Tables 4.1 through 4.4 in the form of four connection matrices. The static connections in the AMG of the data and control edges are shown in Tables 4.1 and 4.2. The position of the initial data and control tokens is shown in Tables 4.3 and 4.4. The desired positions of the processors in the processor queue are specified and placed in the processor queue array as shown in Table 4.5. Next, the node timings are placed in an array with the first position in the array representing the node time for node zero, the second being for node one, and so on. This completes the specification of the example AMG on the testbed and the graph can now be executed.

Execution of the graph begins with the graph manager assigning itself the semaphore to begin token bus operation. The graph manager then checks the graph to find an enabled node. This is done by comparing the first row in Table 4.1 to the first row in Table 4.4 and the first row in Table 4.2 to the first row in Table 4.3. If the two pairs of rows have the same entries (tokens) then that particular node, corresponding to the row index, is enabled with all data and control tokens and is ready to be fired. After finding an enabled node, the graph manager assigns a PE for the execution of the node and updates the graph by consuming all data tokens and generating all necessary control tokens. A PE is assigned to a node operation by shifting all the rows in Table 4.5 to the left by one bit. The processor on top of the queue, indicated by the processor mask that has a "1" shifted out, is assigned to that particular node

operation. The pointer mask is used to monitor the last position in the queue. The graph manager appends the processor ID and the node number to the "Fire" message and places the complete command word on the bus. This continues until all enabled nodes have been fired. When all enabled nodes have been fired, the graph manager releases the semaphore to the next PE in logical sequence. If this PE has been assigned to a node operation, and is finished with that node operation, it puts a "Done" message with its ID and the node number of the executed node. When the graph manager reads the "Done" message it updates the graph to indicate that the node operation is complete. The graph manager also updates the processor queue by copying the value of the pointer mask into the processor mask corresponding to the PE that just completed a node operation. If the PE has not completed the node operation it passes on the semaphore with the ID of the next PE in logical sequence.

Upon reaching a terminating condition, execution of the graph is terminated. The terminating condition, usually the completion of a number of iterations, is specified within the graph manager routine. The graph manager also concerns itself with data collection of execution information for the creation of a "Fire Data Time" (FDT) file. This file is in a format compatible with the ATAMM Analyzer [JONES90] that helps to analyze the performance of an example graph on the testbed.

The graph manager and the PEs contain an initialization and synchronization procedure. The graph manager communication layer is initialized with the semaphore thus giving it control of the bus. The PEs are initialized to start in the idle state. The graph manager then proceeds with task scheduling and message passing in a noncontending and deadlock free manner.

5.2 Timing Measuremnets

The testbed is designed for operation in a manner that is transparent to the user. The user provides the dataflow graph and a node delay that is

equivalent to the amount of time that a node operation takes. Node delays were used instead of actual node code and data for testing the AMOS structure. Along with the graph information node timings are placed in an array and distributed among the PEs. When a graph is executed, the results are reported in the form of a FDT file for analysis using the ATAMM Analysis Tool [JONES90].

5.2.1 FDT Evaluation

Modification of the testbed for timing measurements is done in a manner such that the execution patterns in the form of a FDT(Fire Data Time) file can directly be analyzed by the ATAMM Analysis Tool. The FDT file contains a list of information pertaining to each AMOS broadcast event, in order of occurrence, which provides a means of evaluating system performance and graph execution. Of particular interest is the recording of the

1. time tagging of the firing of a node; and
2. time tagging of a "Done" message.

The graph manager is responsible for the firing of a node and the update of the graph when a "Done" message is reported. Hence the graph manager events are the reference for which AMOS communication events are time-tagged. The position of the time marks in the state diagram for the graph manager is shown in Figure 5.1. A monitor program resident on the host PC of the graphmanbager is used to supervise the execution behavior of the graph, and to collect FDT file information.

5.2.2 Monitor Program

The monitor program is essentially a serial communication program, written in C, that extracts information from messages sent to it by the microcontroller along the serial port and time stamps the messages.

Graph execution is started by the monitor program that sends a message to the graph manager to begin execution. The graph manager finds an enabled node and sends a message on the bus to the PE scheduled for execution of the node. The graph manager also sends a message to the monitor program with

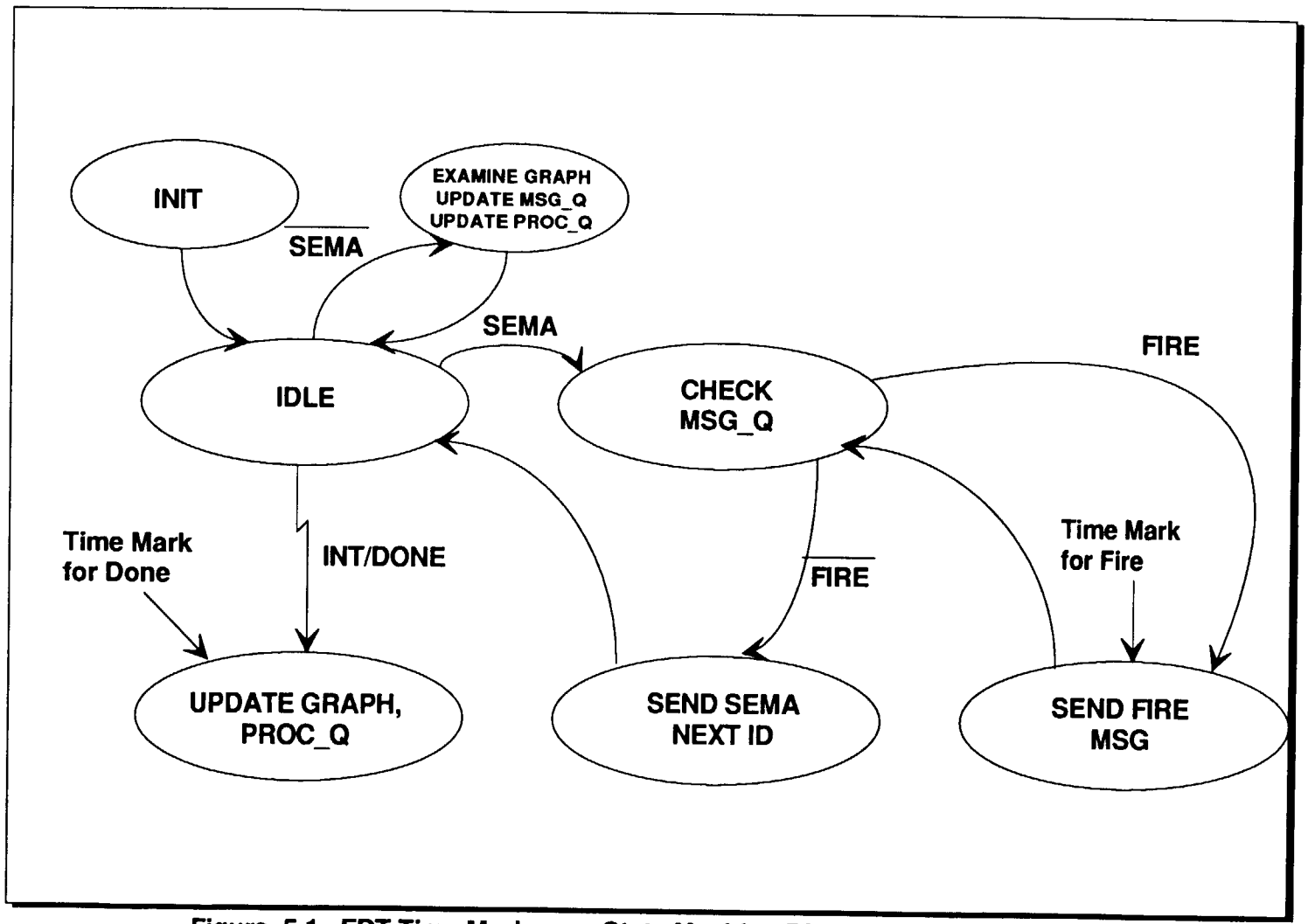


Figure 5.1. FDT Time Marks on State Machine Diagram for Graph Manager.

node number, processor ID and command (Fire). The monitor program time stamps the message and stores it along with an iteration count. When the graph manager receives a "Done" message it sends a message to the monitor program with node number, processor ID, and command(Done). The monitor program time stamps this message and stores it.

When a terminating condition is reached, such as a final iteration count, collection of FDT data is completed. The monitor program then converts the FDT file data into the format specified for the analysis tool. The file format is shown in Table 5.1.

5.3 Verification Experiment

Experiments to demonstrate dynamic scheduling and timing measurements for various examples have been executed on the testbed. An experiment begins with an AMG, which is specified in the form of data structures mentioned in chapter three. The node times are also specified and distributed to each PE. The execution of the graph results in the generation of the FDT file which forms the input to the ATAMM analysis tool. The output of the analysis tool is a graphical representation of the execution pattern of the graph.

For brevity a representative experiment demonstrating the maximum capacity of the testbed is reported herein. Details of other experiments may be found in [SASTRY94].

5.3.1 Eight node example

An eight node AMG that requires a maximum of three processors is used to explore the full capabilities of the testbed. The AMG for the eight node example is presented in Figure 5.2 and the theoretical TGP is shown in Figure 5.3. From the TGP the TBO for the graph is, ideally, 3000 ms. TBO in this case is determined by the recursive circuit node zero-node one-node four. TBIO for this graph is calculated to be 6000 ms. The FDT file from the execution of the graph is shown in Figure 5.4(a).

The analyzer output clearly demonstrates the full capability of the

Table 5.1 FDT File Format as Input to ATAMM Analysis Tool

Time at which the FDT event occurred	Event Type	Node Number	Color (Simplex)	Processor ID Number	Iteration Number
(Milli-seconds)	One of the following: Fire node Done node	[NODE#]	Always 1	[PID_#]	[Iter.#]

Note: FDT events are one of the following:

Fire node : Indicates time at which a node initiated execution.

Done node : Indicates time at which a node completed execution.

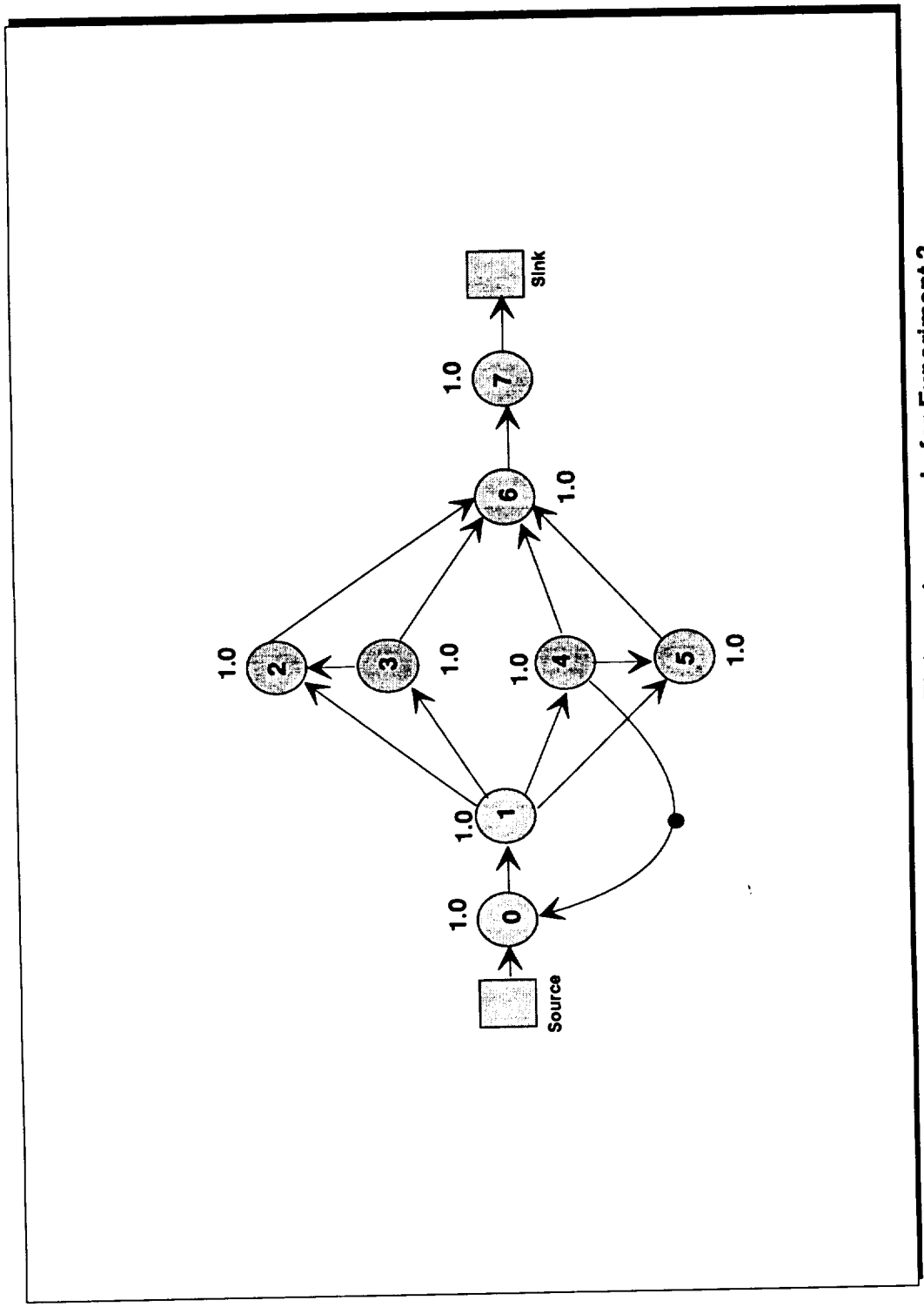


Figure 5.2. AMG for the Eight node example for Experiment 3.

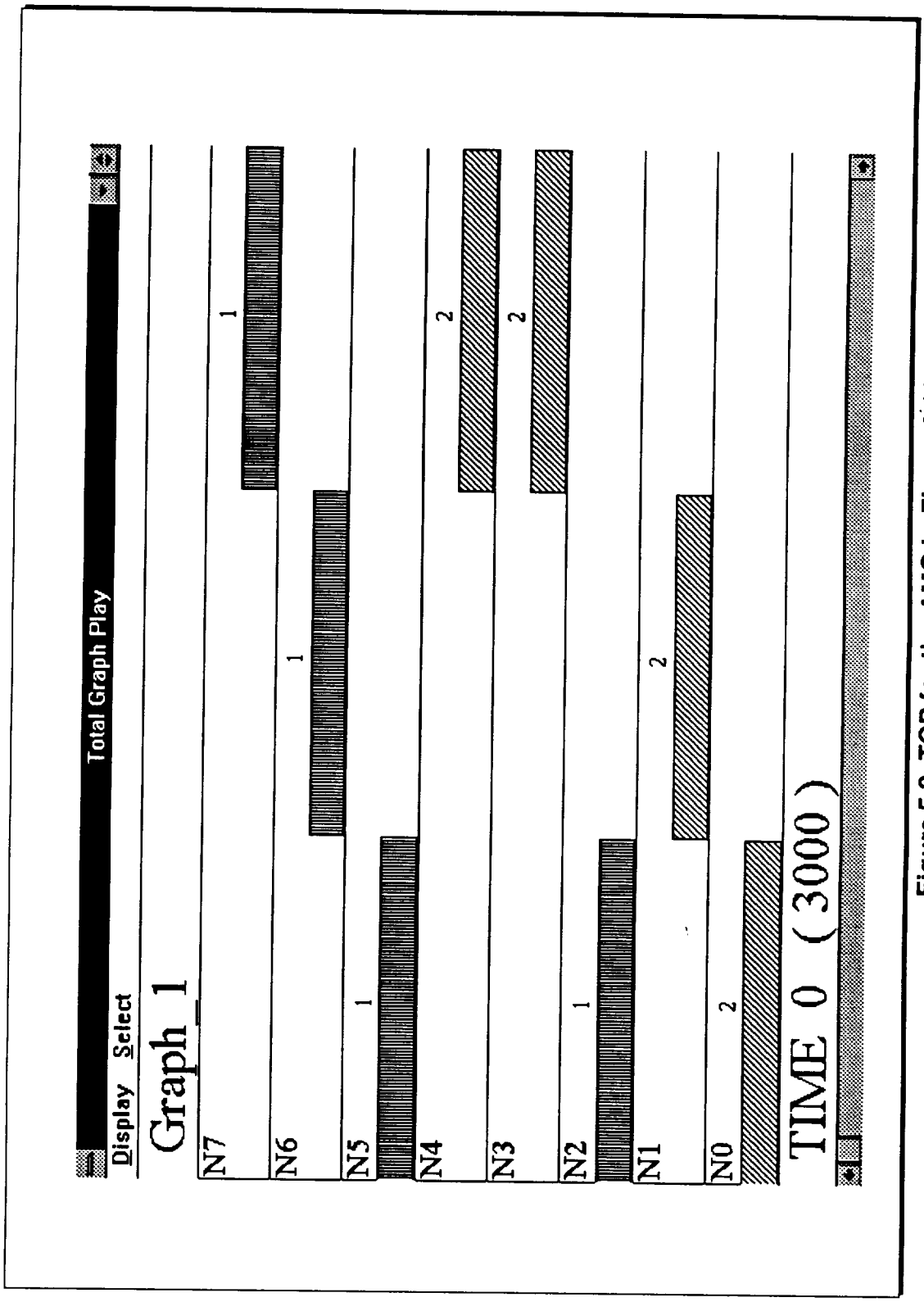


Figure 5.3. TGP for the AMG in Figure 4.11.

testbed to execute an eight node graph that uses three processors. The TBO from the analyzer output is equal to 2995 ms and the TBIO is equal to 5967 ms. The steady state behavior of the graph is in accordance with the behavior predicted by the Design Tool. A section of the analyzer output (Figure 5.4(a) is expanded to one TGP frame and is shown in Figure 5.4(b). The steady state behavior of the graph and the effect of delays in the "Done" messages arriving at the graph manager are clearly shown in the TGP frame. For example, node five for iteration two is enabled at the completion of iteration two of node four, but it cannot be "Fired" until iteration one for node seven is complete because two PEs are being used for the execution of iteration two for node two and iteration three for node zero. These delays are due to the measurement resolution limitation of 55 ms. which is discussed in section 5.4.3,

5.4 Timing Measurements

Interprocessor communication is achieved in the testbed using a token bus network as discussed in Chapter Four. Of interest is the overhead incurred by AMOS using the token bus interconnection network and the overhead incurred in the graph management.

5.4.1 Time to Pass Messages Around the Logical Ring

The time required to pass messages around the logical ring formed by the token bus network was measured using an oscilloscope connected to the strobe lines of one PE. With only two PEs on the bus, the first measurement was made with one PE writing to the bus and another reading the bus. The time for a message to be passed from one PE to another was measured to be 20 microseconds. With three PEs connected the message passing time increased to 42 microseconds. With four PEs, the time required to pass messages was found to be 62 microseconds. The relationship of the increase in message passing time with the number of PEs is shown in Figure 5.5 and appears to be linear.

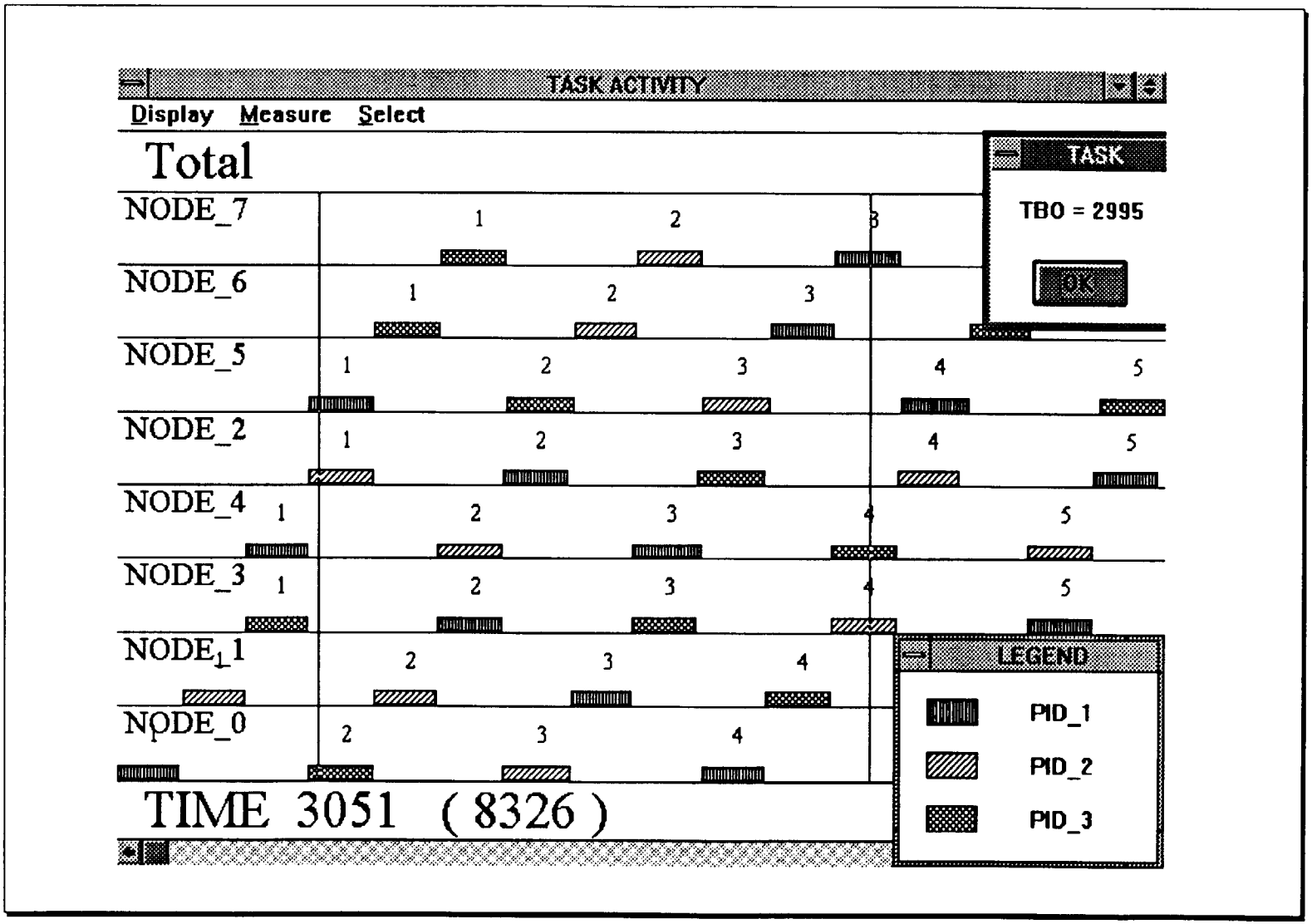


Figure 5.4(a). Analyzer output for the AMG in Figure 4.11.

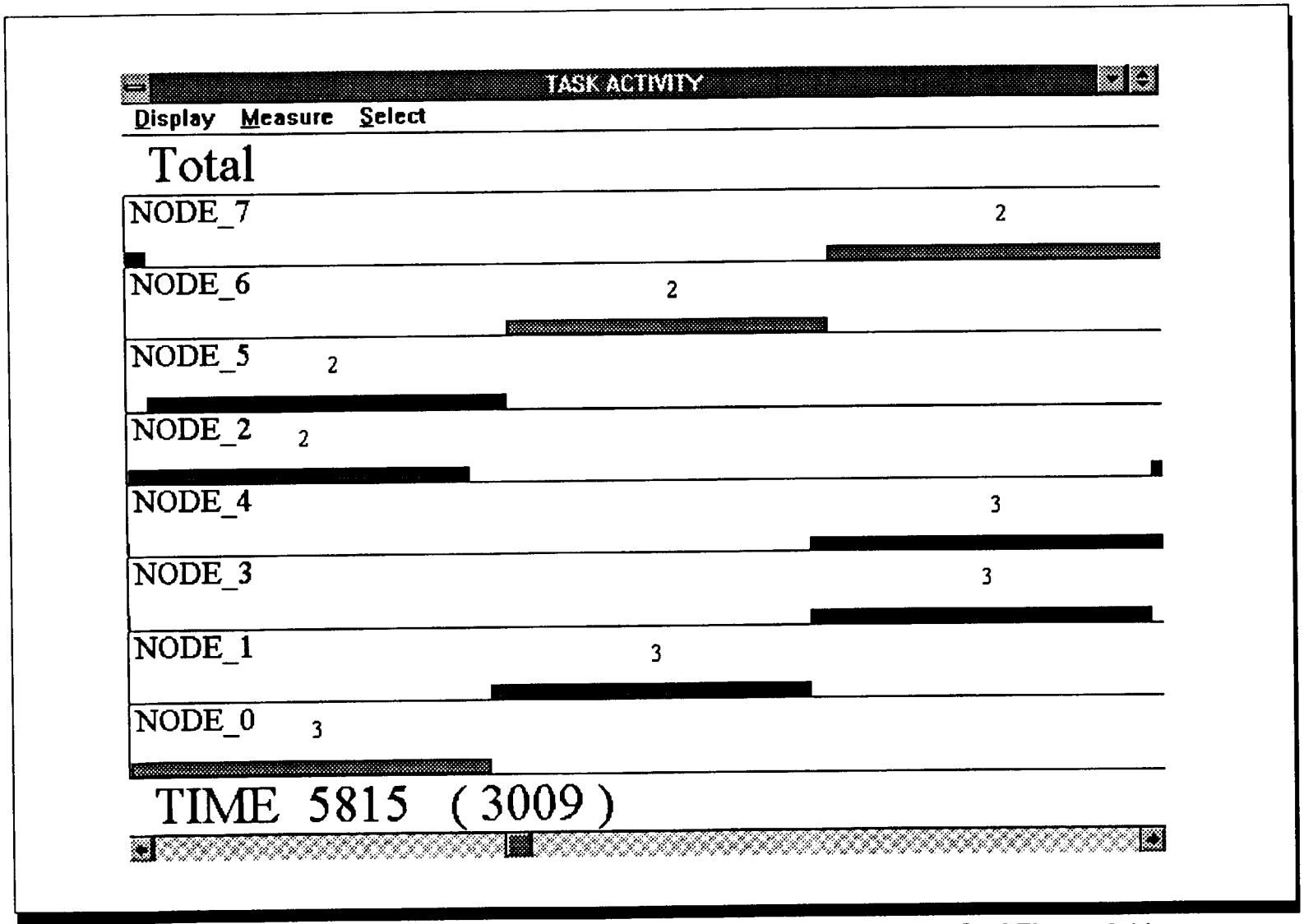


Figure 5.4(b). Analyzer Output for One TBO frame of the AMG of Figure 4.11.

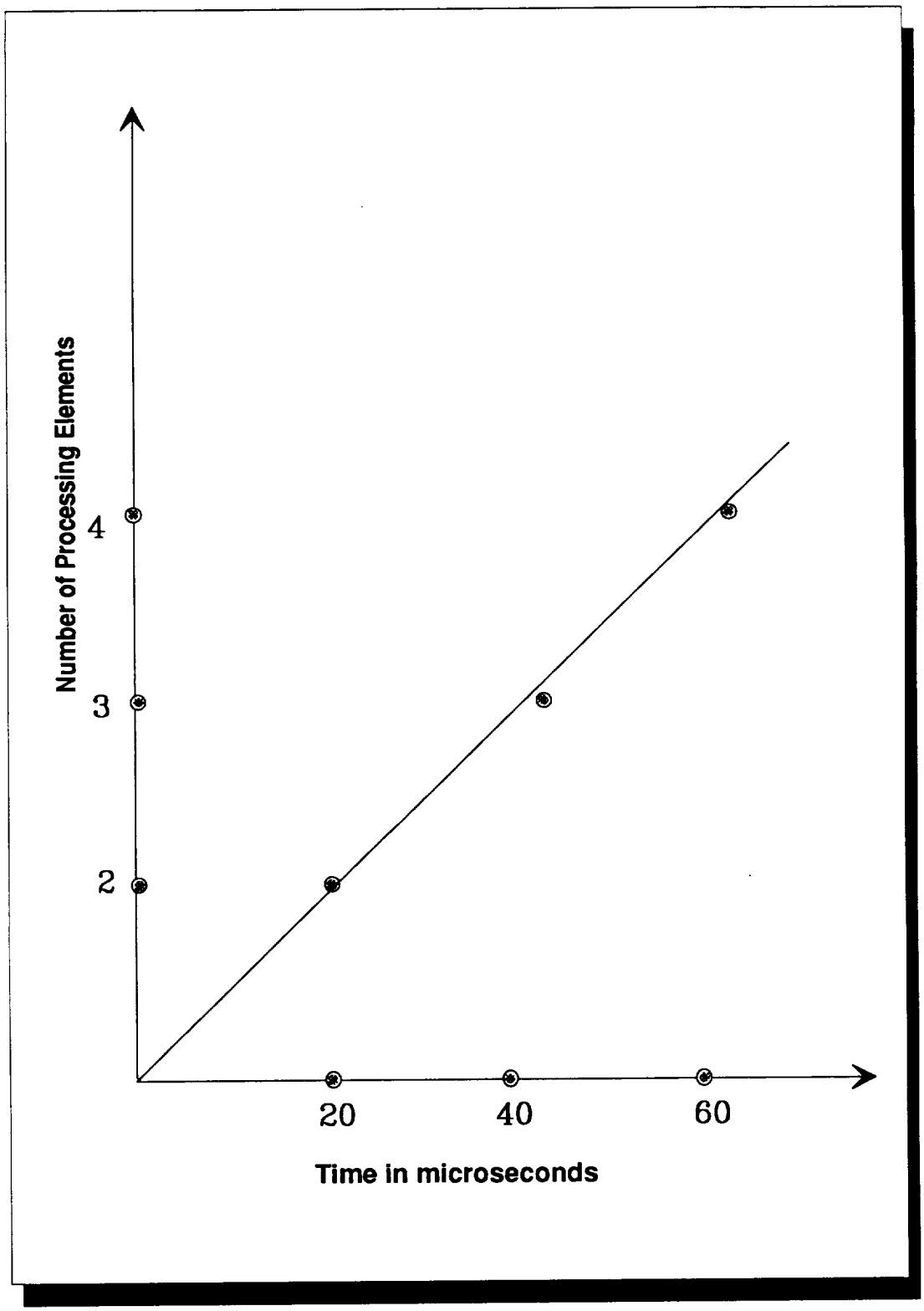


Figure 5.5 Overhead increase with increase in number of PEs.

5.4.2 Software Overhead for the Graph Manager and PEs

Overhead introduced by the graph manager for centralized graph management was measured by using the 68HC11's free running timing counter. The value of the counter was read at the start and the end of a timing measurement. The difference in the counter values was recorded and the time calculated from this difference since the counter is incremented every clock cycle, ie, every 500 nanoseconds.

The flow diagram for the operation of the graph manager of redrawn in Figure 5.6 along with the timing measurements for the different sections of the graph manager code. Recalling that all messages are interrupt driven, timings that are shown include the time required to service the interrupt service routine for the messages and the time required for the main routine to respond to the soft messages passed to it from the interrupt service routine. Each PE has a maximum message queue length of one, since the only message the PE to report is a "Done". The message queue for the graph manager can have three messages in it at any given time. This upper bound on the length of the message queue is due to the testbed's limitation of three PEs. Thus, not more than three nodes can be fired at any given time. The time shown for the graph manager to check the message queue and send a "Fire" command is equal to 142 microseconds. This time is only for the graph manager to send one "Fire" command and send the semaphore to the next PE in the logical ring. The time required for the graph manager to send two and three "Fire" messages is shown in an extensive list of timing measurements for the graph manager and PE operation in Table 5.2. It should be noted that these timings stay consistent with repeated measurements, with an error of +/- 10 microseconds for the time for the graph manager to examine the graph and generate a "Fire" message. This is due to the graph manager taking a different amount of time for generating a "Fire" message for different node numbers as it would take more time to examine the graph and find node seven enabled than it would for node zero. An important observation to be made is that the overhead required

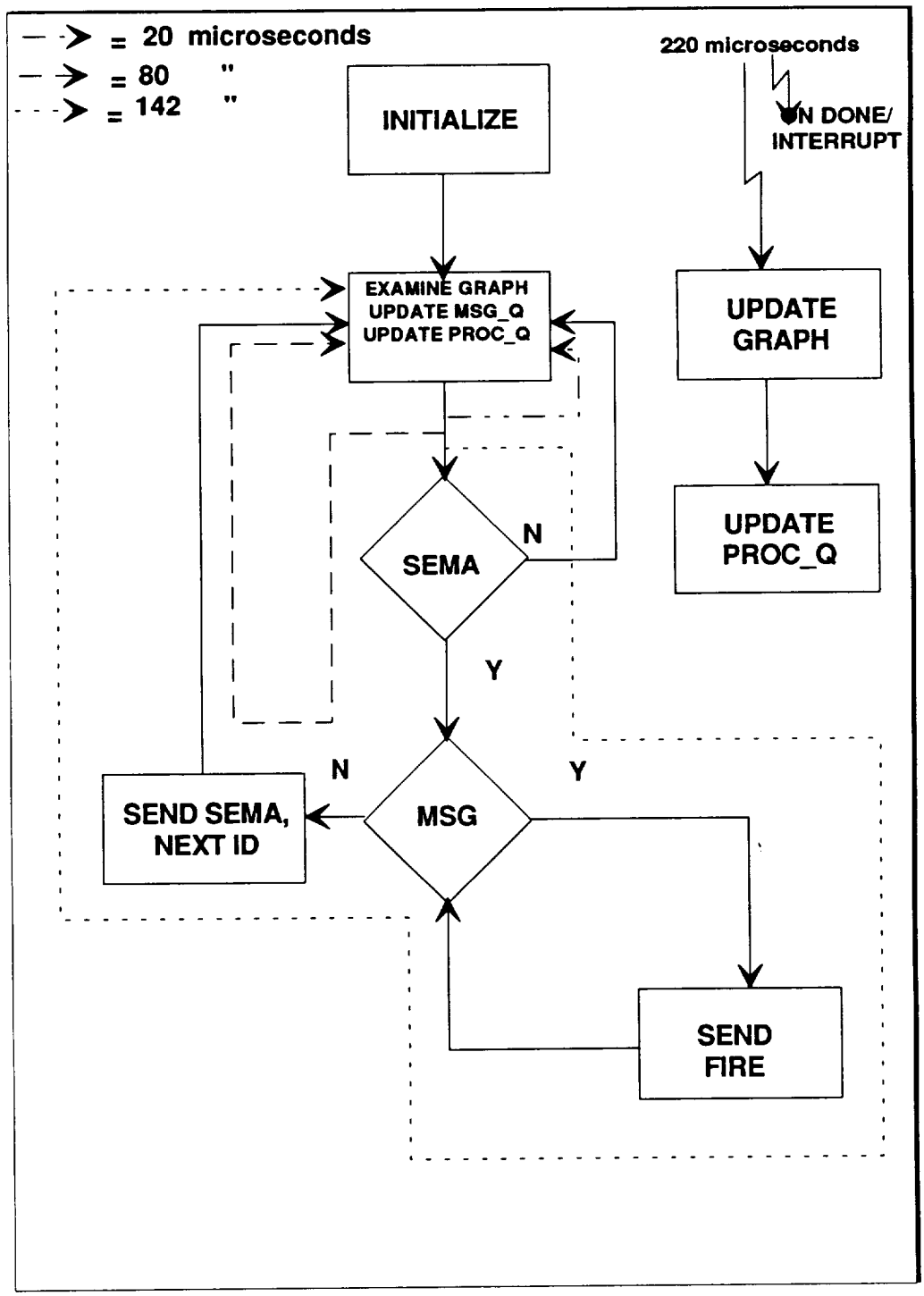


Figure 5.6. Timing measurements for graph manager operation.

Table 5.2 Event for timing measurement.	Time in micro seconds.
Time for a PE or Graph manager to check the message on the bus for its ID.	20
Time for a PE or graph manager to release the semaphore if it has no messages to send.	85
Time for a PE to send a message, and release the semaphore.	145
Time for the graph manager to send a "Fire", check its message queue, and release the semaphore.	142
Time for the graph manager to send two "Fire" commands, check its message queue and release the semaphore.	225
Time for the graph manager to send three "Fire" commands, check its message queue, and release the semaphore.	308
Time for the graph manager to examine the graph, find an enabled node, assign a PE to it, and update the graph.	225
Time for the graph manager to update the graph and processor queue on receiving a "Done".	220

by the graph manager is not significant when compared to the node times considered for the experiments.

5.4.3 Measurement Imprecision

As described in Section 5.3.2, the monitor program resident in the host PC of the graph manager is responsible for time stamping the messages sent to it by the graph manager. Time is marked using the system clock of the PC using a C function that gets the current time of the system. The system clock is updated 18.2 times a second or approximately every 55 ms via an interrupt. Thus, the precision of the time measurement for the creation of the FDT file is approximately +/- 55 ms.

CHAPTER SIX
CONCLUSIONS

6.1 Introduction

The description of two very different and contrasting embodiments of AMOS have been the subject of this report. On one hand is a high level distributed LAN connected testbed employing static scheduling and distributed graph management. The other embodiment employed microcontrollers in a token bus using a centralized graph manager and dynamic scheduling. Significant attrributes of the two highly contrasted embodiments of AMOS are outlined in the following sections.

6.2 LAN Connected Distributed AMOS

Research has addressed the design, implementation and evaluation of a LAN based multicomputing test bed based on static task scheduling strategy for the ATAMM Multicomputer Operating System. Features of the testbed and implementation are summarized below.

1. Hardware for the testbec consisted of six PC-AT clones, interconnected by a peer to peer ethernet connection. RAM disk space reduced server overhead.
2. The distributed AMOS testbed successfully demonstrated various cyclo-static scheduling policies including cyclo-static, block cyclo static and static schedules.
3. Demonstration AMGs included such features as self loops, forwarded tokens, buffers on CMG control arcs. In additional exercises not reported herein, the testbed successfully executed AMGs requiring multiple instantiations of nodes. In all cases the performance results compared

favorably with performance predicted by the ATAMM Design Tool.

4. The execution of an eight node AMG on six processors demonstrated the upper operating limits of the testbed. Communication events occurring in the testbed were limited to the 55 ms. resolution of the internal PC timer.

5. The development of a truely distributed AMOS is significant in that graph management was performed by each individual graph manager's local knowledge of the graph requirements. It should also be noted that this was the first successful attempt to implement a purely distributed ATAMM based operating system.

6. Software for driving the distributed and related data structures were very modest,consisting of about one thousand lines of C code.

6.3 Microcontroller Based AMOS Testbed

Research has addressed the design, implementation and evaluation of a microcontroller based dynamic task scheduling strategy for the ATAMM Multicomputer Operating System. Features of the testbed and implementation are enumerated below.

1. The testbed consisted of 4 68HC11 based microcontroller platforms of which one was dedicated as a central graph manager and the other three served as the processing elements.

2. A dynamic task scheduling strategy for distributed processing using the ATAMM computing paradigm was implemetnted on the centralized graph manager.

3. The testbed was developed around a message passing model which is inherent to the AMOS structure.

4. The interconnection network employed was a token bus to provide a contention free and deterministic basis for message passing. Contention free task scheduling also implies that the testbed is deterministic and well suited for the real time algorithms..

5. Analysis of dataflow graphs executed on the testbed can be done by the

use of the FDT file that is created when a graph is executed.

6. Memory requirements are modest, including a little over .5 Kbyte for the graph manager and communciation layer.

7. The execution behavior of example graphs wereo predicted using the ATAMM Design Tool. The results obtained from FDT analysis were found to be in accordance with the predicted behavior and provided verification of the testbed for execution of data flow graphs in the ATAMM context.

8. A timing resolution limitation of 55 ms was noted in the monitoring of the testbed operation. However, this did not significantly alter the predicted behavior of the testbed, given that the node times were in the order of seconds.

9. Communication overhead introduced by the use of the token bus model was investigated by real time monitoring. These times, on the order of tens to hundreds of microseconds would be suitable for node times in the order of single digit milliseconds. Note that much larger node times on the order of seconds were used in the test examples.

10. The use of a generic eight bit microcontroller was found to be sufficient to implement the control structure for an ATAMM based testbed. The net amount of program memory available on each microcontroller was 768 bytes, of which 742 bytes were used for the 68HC11 that contained the software for the graph manager and 560 bytes were used for each of the PEs.

REFERENCES

[AGERWALA82] Tilak Agerwala and Arvind, "Data Flow Systems," *IEEE Computer*, February 1982, pp.10-14.

[ANDREWS93] Asa M. Andrews, CTA Incorporated, VA, *Graph Entry Tool*, Version 2.5.17, 1993.

[BABB84] R.G. Babb, "Parallel Processing with large grain dataflow techniques," *Computer*, Vol.17, July, 1984.

[COURT92] R.Court, "Real-time Ethernet," *Computer Communications*, Vol.15, No.3, April 1992.

[CHASE87] M.Chase, "A pipelined dataflow architecture for signal processing: The NEC μPD7281," in *VLSI Signal Processing*, New York: IEEE Press, 1984.

[ERCEGOVAC86] Milos D. Ercegovac and Tomás Lang, "General Approaches for Achieving High Speed Computations," *Supercomputers Class VI Systems, Hardware and Software*. S.Fernbach (Editor), Elsevier Science Publishers B.V. (North Holland), 1986, pp.1-28.

[FURTNEY93] Mark Furtney and George Taylor, "Of Workstations and Supercomputers," *IEEE Spectrum*, May 1993, pp.64-68.

[GORSLINE86] Gorsline, G.W. *Computer Organization*, Prentice Hall, Englewood Cliffs, Inc., N.J.

[HENNESSEY90] John L. Hennessey and David A. Patterson, *Computer Architecture : A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., CA 1990.

[HWANG84] Kai Hwang and F.A.Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, NY, 1984, pp.732-768.

[JONES90] Robert L. Jones, III, "Diagnostics Software for Concurrent Processing Computer Systems," M. S. Thesis, Old Dominion University, Norfolk, Virginia, April 1990.

[JONES93] R.L. Jones, III, NASA Langley Research Center, VA, *ATAMM Analysis Tool*, Version 3.1, 1993.

[LAUER79] Lauer, H.C., and R.M Needham. "On the Duality of Operating Systems Structures." Proc. Second International.Symp. Operating Systems, IRIA, Oct. 1978.

[LEE87] Edward Ashford Lee and David G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", *IEEE Transactions on Computers*, Vol. C-36, No.1., January 1987, pp.24-35.

[MIELKE88] Roland R. Mielke, John W. Stoughton and Sukhamoy Som, "Modelling and Optimum Time Performance for Concurrent Processing," NASA Technical Paper 4167, Grant NAG1-683, August 1988.

[MIELKE90] R.R.Mielke, J.W.Stoughton, S.Som, R.Obando, M.Malekpour and B.Mandala, "Algorithm to Architecture Mapping Model (ATAMM) Multicomputer Operating System Functional Specification," NASA Contractor Report 4339, Cooperative Agreement NCC1-136, November 1990.

[PETERSON81] J.L. Peterson, "Petri Net Theory and the Modelling of Systems," Prentice Hall, Englewood Cliffs, NJ, 1981.

[RASMUSSEN87] R.D. Rasmussen, G.S. Bolotin, N.J. Dimopoulos, B.F. Lewis, and R.M. Manning, "Advanced General Purpose Multicomputer for Space Applications," *Proceedings of the 1987 International Conference on Parallel Processing*, University Park, PA, USA, August 17-21, 1987, pp.54-57.

[RISHE91] N.Rishe, D.Tal, S.Navathe and S.Graham, "On Parallel Architectures," *Parallel Architectures*, IEEE Computer Society Press, Los Alamitos, CA, 1991.

[ROY93] Sudeepto Roy, "Cyclo-Static Scheduling of Large Grain Data Flow Algorithms on a Local Area ATAMM Multicomputing Testbed," Master's Thesis, Old Dominion University, Norfolk, Virginia, December, 1993.

[SASTRY94] Sudhir Sastry,"Dynamic Task Scheduling for the ATAMM Multicomputer Operating System Using Embedded Firmware on Microcontrollers," Master's Thesis, Old Dominion University, Norfolk, Virginia, January, 1994.

[SCHWARTZ85] D.A.Schwartz and T.P.Barnwell III, "Cyclo-Static Multiprocessor Scheduling for the Optimal Realization of Shift-Invariant Flow Graphs", *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, Tampa, Florida, 1985.

[SOM88] Sukhamoy Som, "Performance Modelling and Enhancement for ATAMM Data Flow Architectures," Ph.D. Dissertation, Old Dominion University, Norfolk, Virginia, May 1989.

[SOM90] S.Som, B.Mandala, R.R.Mielke and J.W.Stoughton, "A Design Tool for Computations in Large Grain Real Time Dataflow Architectures," *Proceedings of the IEEE Southeastcon '90*, New Orleans, LA, April 1990.

[SOM93] S.Som, R.Obando, R.R.Mielke and J.W.Stoughton, "ATAMM: A Computational Model for Real-Time Data Flow Architectures," *International Journal of Mini and Microcomputers*, Vol.15, No.1, 1993, pp.11-22.

[STALLINGS91] Stallings, W. *Computer Architecture*, Macmillan Publishers. 1991.

[STOUGHTON86] J.W. Stoughton and R.R. Mielke, "Petri-Net Model for Concurrent Processing of Complex Algorithms," *Proceedings of Government Microcircuit Applications Conference*, San Diego, California, November 1986.

[STOUGHTON88] J.W. Stoughton and R.R. Mielke, "Strategies for Concurrent Processing of Complex Algorithms in Data Driven Architectures," NASA Technical Paper 181657, Grant NAG1-683, February 1988.

[STOUGHTON93] Private conversations and technical exchanges with Dr.J.W.Stoughton.

[TANENBAU92] Andrew S.Tanenbaum, M. Frans Kaashoek and Henri E. Bal, "Parallel Programming Using Shared Objects and Broadcasting", *IEEE Computer*, August 1992, pp.10-12.

[TYMCHYSHN88] William Robert Tymchyshyn, "ATAMM Multicomputer System Design," M. S. Thesis, Old Dominion University, Norfolk, Virginia, August 1988.